Curso Programación Android



Salvador Gómez Oliver

WWW.SGOLIVER.NET

Versión 2.0 // Noviembre 2011

ÍNDICE DE CONTENIDOS

PRÓLOGO

Hay proyectos que se comienzan sin saber muy bien el rumbo exacto que se tomará, ni el destino que se pretende alcanzar. Proyectos cuyo único impulso es el día a día, sin planes, sin reglas, tan solo con el entusiasmo de seguir adelante, a veces con ganas, a veces sin fuerzas, pero siempre con la intuición de que va a salir bien.

El papel bajo estas líneas es uno de esos proyectos. Nació casi de la casualidad allá por 2010. Hoy estamos viendo como acaba el 2011 y sigue más vivo que nunca.

A pesar de llevar metido en el desarrollo para Android casi desde sus inicios, en mi blog [sgoliver.net] nunca había tratado estos temas, pretendía mantenerme fiel a su temática original: el desarrollo bajo las plataformas Java y .NET. Surgieron en algún momento algunos escarceos con otros lenguajes, pero siempre con un ojo puesto en los dos primeros.

Mi formación en Android fue en inglés. No había alternativa, era el único idioma en el que, por aquel entonces, existía buena documentación sobre la plataforma. Desde el primer concepto hasta el último tuve que aprenderlo en el idioma de Shakespeare. A día de hoy esto no ha cambiado mucho, la buena documentación sobre Android, la buena de verdad, sigue y seguirá aún durante algún tiempo estando en inglés, pero afortunadamente son ya muchas las personas de habla hispana las que se están ocupando de ir equilibrando poco a poco esta balanza de idiomas.

Y con ese afán de aportar un pequeño granito de arena a la comunidad hispanohablante es como acabé decidiendo dar un giro, quien sabe si temporal o permanente, a mi blog y comenzar a escribir sobre desarrollo para la plataforma Android. No sabía hasta dónde iba a llegar, no sabía la aceptación que tendría, pero lo que sí sabía es que me apetecía ayudar un poco a los que como yo les costaba encontrar información básica sobre Android disponible en su idioma.

Hoy, gracias a todo vuestro apoyo, vuestra colaboración, vuestras propuestas, y vuestras críticas (de todo se aprende) éste es un proyecto con más de un año ya de vida. Más de 250 páginas, más de 40 artículos, y sobre todo cientos de comentarios de ánimo recibidos.

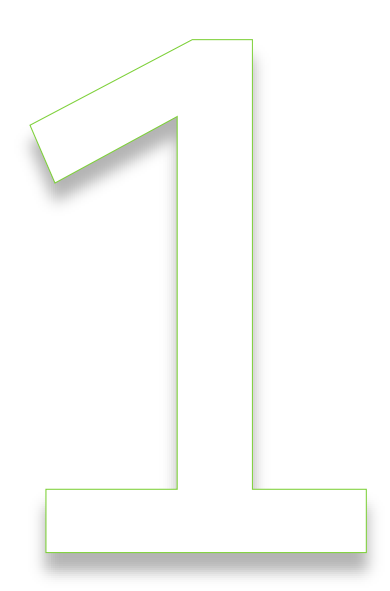
Y este documento no es un final, es sólo un punto y seguido. Este libro es tan solo la mejor forma que he encontrado de mirar atrás, ordenar ideas, y pensar en el siguiente camino a tomar, que espero sea largo. Espero que muchos de vosotros me acompañéis en parte de ese camino igual que lo habéis hecho en el recorrido hasta ahora.

Muchas gracias, y que comience el espectáculo.

¿A QUIÉN VA DIRIGIDO ESTE LIBRO?

Este manual va dirigido a todas aquellas personas interesadas en un tema tan en auge como la programación de aplicaciones móviles para la plataforma Android. Se tratarán temas dedicados a la construcción de aplicaciones nativas de la plataforma, dejando a un lado por el momento las aplicaciones web. Es por ello por lo que el único requisito indispensable a la hora de utilizar este manual es tener conocimientos bien asentados sobre el lenguaje de programación Java y ciertas nociones sobre aspectos básicos del desarrollo actual como la orientación a objetos.

LICENCIA



Conceptos Básicos

I. Conceptos Básicos

Entorno de desarrollo Android

En este apartado vamos a describir los pasos básicos para disponer en nuestro PC del entorno y las herramientas necesarias para comenzar a programar aplicaciones para la plataforma Android.

No voy a ser exhaustivo, ya existen muy buenos tutoriales sobre la instalación de Eclipse y Android, incluida la documentación oficial de la plataforma. Además, si has llegado hasta aquí quiero suponer que tienes unos conocimientos básicos de Eclipse y Java, por lo que tan sólo enumeraré los pasos necesarios de instalación y configuración, y proporcionaré los enlaces a las distintas herramientas. Vamos allá.

Paso 1. Descarga e instalación de Eclipse.

Si aún no tienes instalado Eclipse, puedes descargar la última versión (3.7 en el momento de escribir estas líneas) desde este enlace. Recomiendo descargar por ejemplo la versión "*Eclipse IDE for Java Developers*". La instalación consiste simplemente en descomprimir el ZIP en la ubicación deseada.

Paso 2. Descargar el SDK de Android.

El SDK de la plataforma Android se puede descargar desde este enlace (la última versión disponible en el momento de escribir este tema es la r15). Una vez descargado, de nuevo bastará con descomprimir el zip en cualquier ubicación.

Paso 3. Descargar el plugin Android para Eclipse.

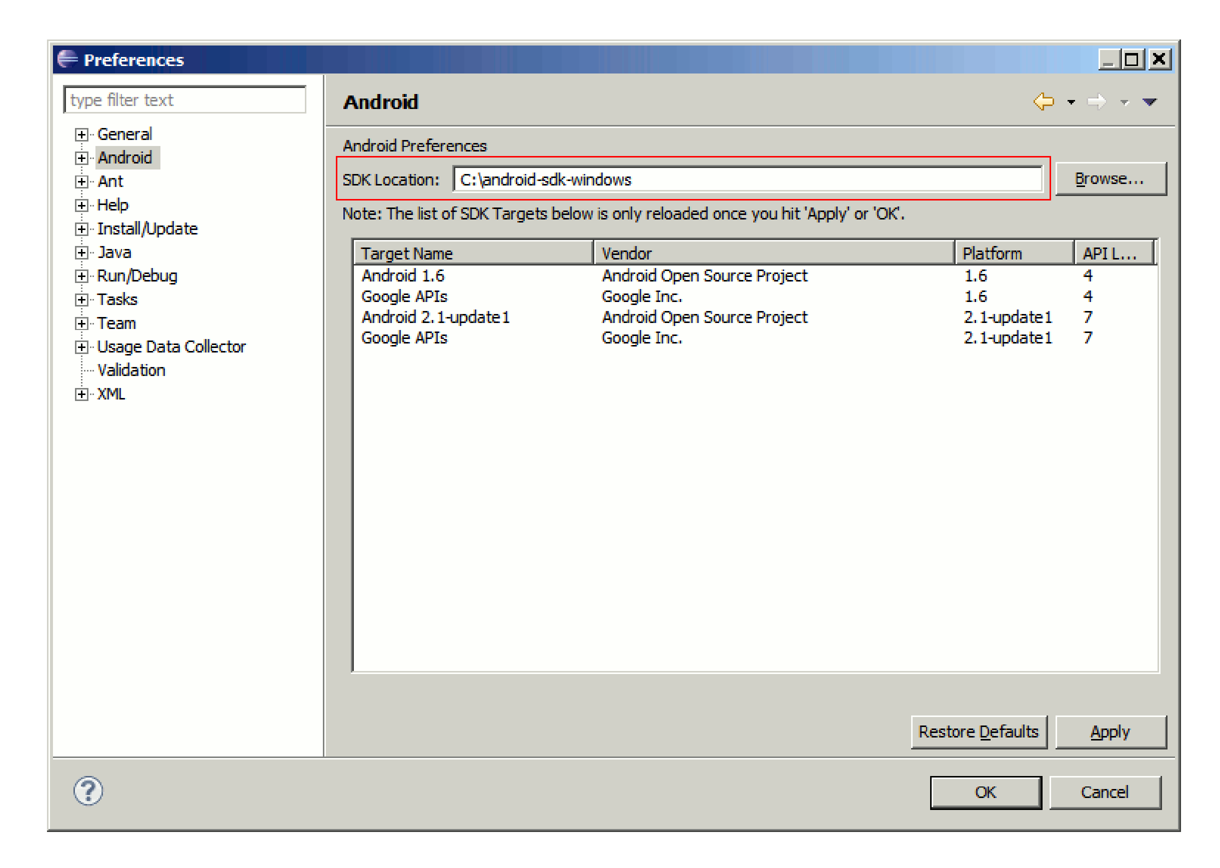
Google pone a disposición de los desarrolladores un plugin para Eclipse llamado *Android Development Tools (ADT)* que facilita en gran medida el desarrollo de aplicaciones para la plataforma. Podéis descargarlo mediante las opciones de actualización de Eclipse, accediendo al menú "*Help / Install new software…*" e indicando la URL de descarga:

```
https://dl-ssl.google.com/android/eclipse/
```

Se debe seleccionar e instalar el paquete completo *Developer Tools*, formado por *Android DDMS* y *Android Development Tools*.

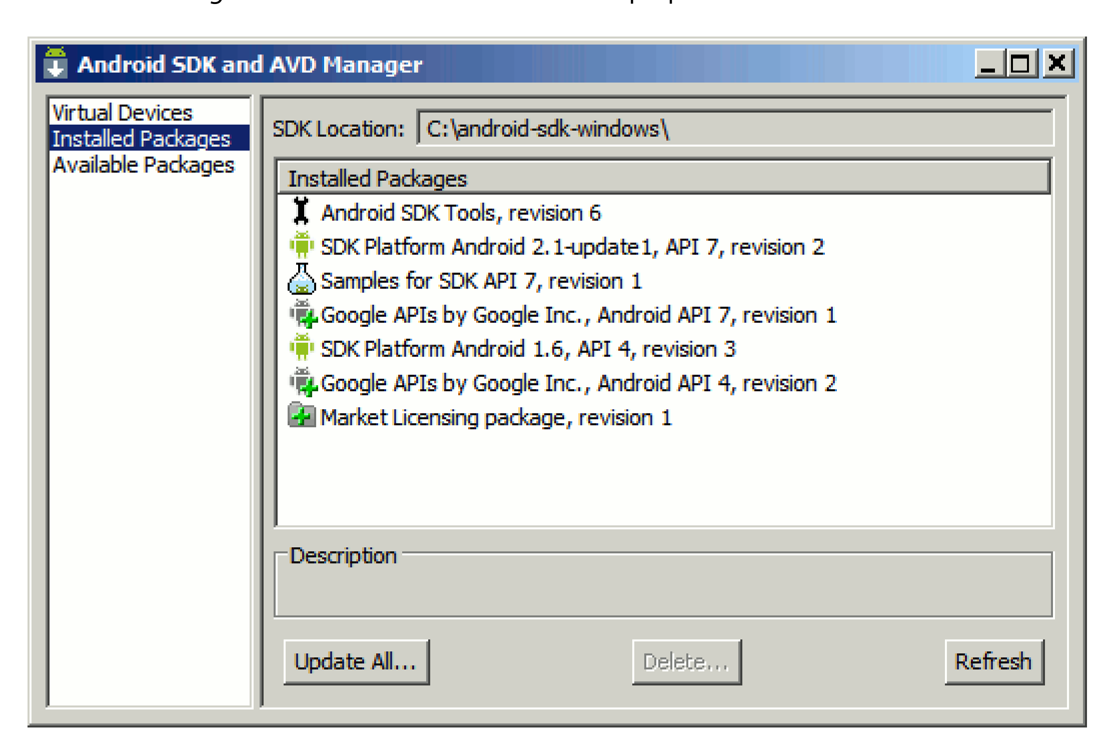
Paso 4. Configurar el plugin ADT.

En la ventana de configuración de Eclipse, se debe acceder a la sección de Android e indicar la ruta en la que se ha instalado el SDK (paso 2).



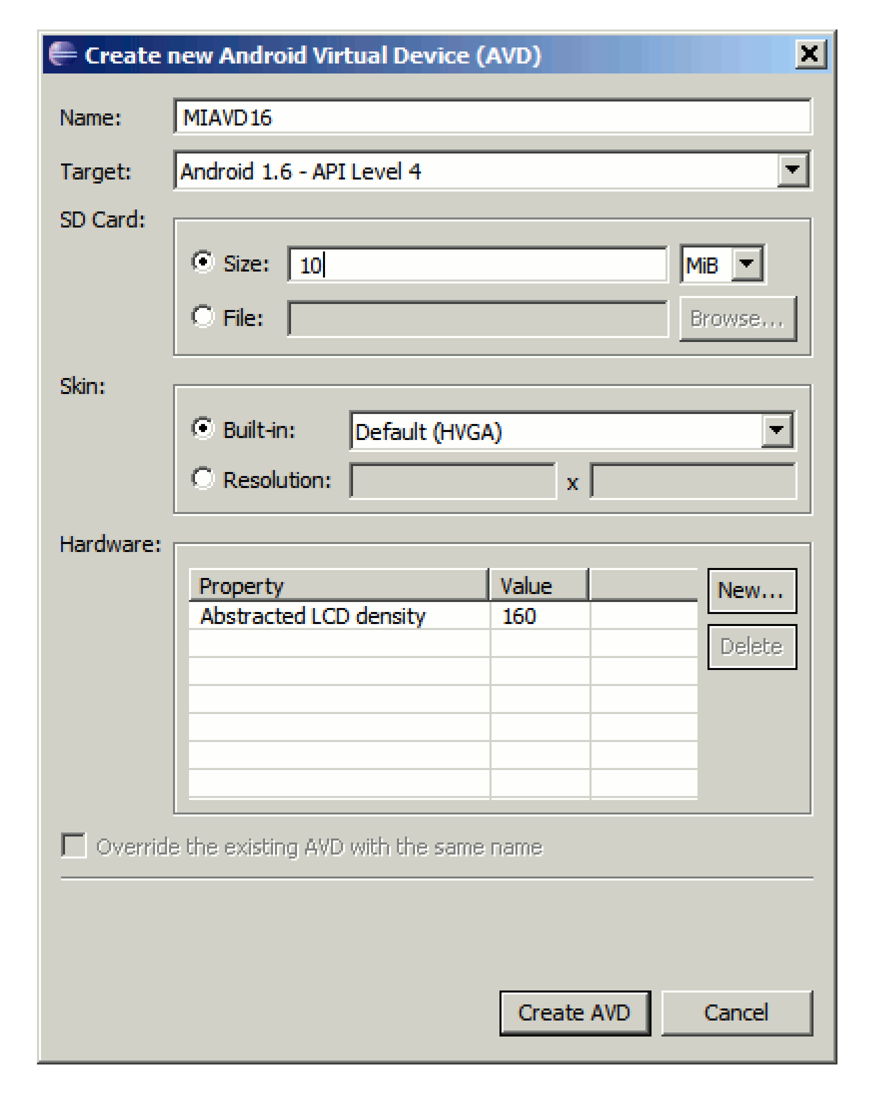
Paso 5. Descargar los targets necesarios.

Además del SDK de Android comentado en el paso 2, también debemos descargar los llamados *SDK Targets* de Android, que no son más que las librerías necesarias para desarrollar en cada una de las versiones concretas de Android. Así, si queremos desarrollar por ejemplo para Android 1.6 tendremos que descargar su target correspondiente. Para ello, desde Eclipse debemos acceder al menú "*Window / Android SDK and AVD Manager*", y en la sección *Available Packages* seleccionar e instalar todos los paquetes deseados.



Paso 6. Configurar un AVD.

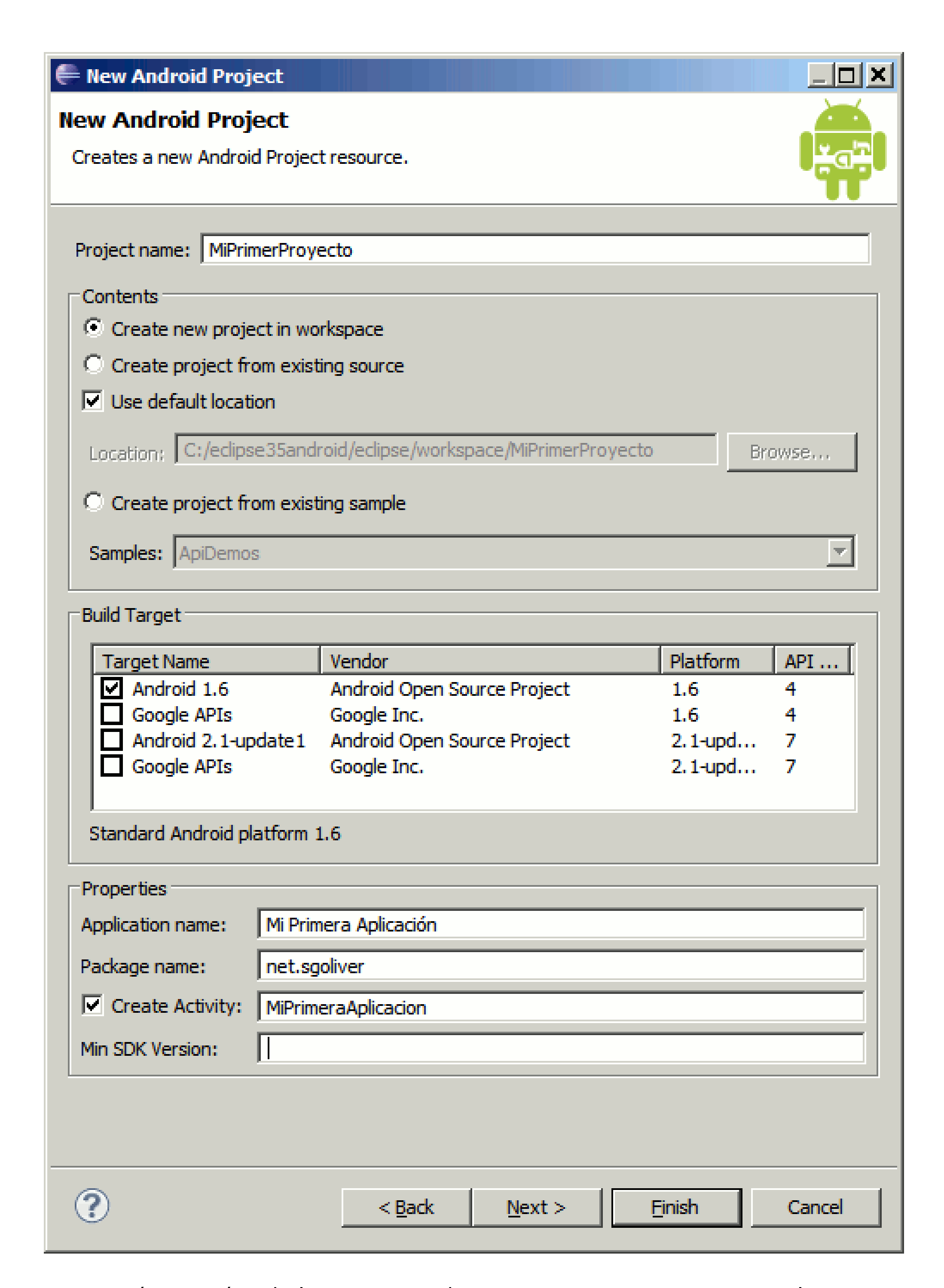
A la hora de probar y depurar aplicaciones Android no tendremos que hacerlo necesariamente sobre un dispositivo físico, sino que podremos configurar un emulador o dispositivo virtual (*Android Virtual Device*, o AVD) donde poder realizar fácilmente estas tareas. Para ello, volveremos a acceder al AVD Manager, y en la sección *Virtual Devices* podremos añadir tantos AVD como se necesiten (por ejemplo, configurados para distintas versiones de Android). Para configurar el AVD tan sólo tendremos que indicar un nombre descriptivo, el target de Android que utilizará, y las características de hardware del dispositivo virtual, como por ejemplo su resolución de pantalla, el tamaño de la tarjeta SD, o la disponibilidad de GPS.



Y con este paso ya estamos preparados para crear nuestro primer proyecto para Android.

Paso 7. ¡Hola Mundo! en Android.

Creamos un nuevo proyecto de tipo *Android Project*. Indicamos su nombre, el target deseado, el nombre de la aplicación, el paquete java por defecto para nuestras clases y el nombre de la clase (*activity*) principal.



Esta acción creará toda la estructura de carpetas necesaria para compilar un proyecto para Android. Hablaremos de ella más adelante.

Para ejecutar el proyecto tal cual podremos hacerlo como cualquier otro proyecto java configurando una nueva entrada de tipo *Android Applications* en la ventana de *Run Configurations*. Al ejecutar el proyecto, se abrirá un nuevo emulador Android y se cargará automáticamente nuestra aplicación.

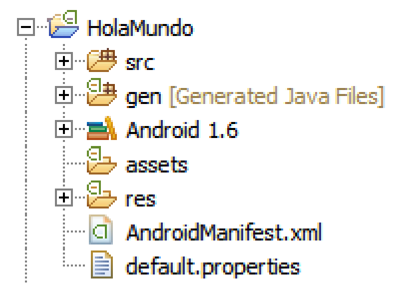


Estructura de un proyecto Android

Para empezar a comprender cómo se construye una aplicación Android vamos a echar un vistazo a la estructura general de un proyecto tipo.

Cuando creamos un nuevo proyecto Android en Eclipse se genera automáticamente la estructura de carpetas necesaria para poder generar posteriormente la aplicación. Esta estructura será común a cualquier aplicación, independientemente de su tamaño y complejidad.

En la siguiente imagen vemos los elementos creados inicialmente para un nuevo proyecto Android:



En los siguientes apartados describiremos los elementos principales de esta estructura.

Carpeta /src/

Contiene todo el código fuente de la aplicación, código de la interfaz gráfica, clases auxiliares, etc. Inicialmente, Eclipse creará por nosotros el código básico de la pantalla (`Activity`) principal de la aplicación, siempre bajo la estructura del paquete java definido.

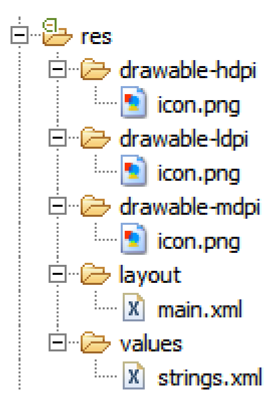


Carpeta /res/

Contiene todos los ficheros de recursos necesarios para el proyecto: imágenes, vídeos, cadenas de texto, etc. Los diferentes tipos de recursos de deberán distribuir entre las siguientes carpetas:

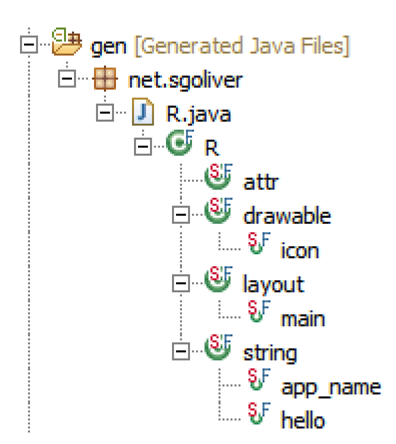
Carpeta	Descripción
`/res/drawable/`	Contienen las imágenes de la aplicación. Para utilizar diferentes recursos dependiendo de la resolución del dispositivo se suele dividir en varias subcarpetas: ➢ `/drawable-ldpi` ➢ `/drawable-mdpi` ➢ `/drawable-hdpi`
`/res/layout/`	Contienen los ficheros de definición de las diferentes pantallas de la interfaz gráfica. Para definir distintos layouts dependiendo de la orientación del dispositivo se puede dividir en dos subcarpetas: ➢ `/layout` ➢ `/layout-land`
`/res/anim/`	Contiene la definición de las animaciones utilizadas por la aplicación.
`/res/menu/`	Contiene la definición de los menús de la aplicación.
`/res/values/`	Contiene otros recursos de la aplicación como por ejemplo cadenas de texto (*strings.xml*), estilos (*styles.xml*), colores (*colors.xml*), etc.
`/res/xml/`	Contiene los ficheros XML utilizados por la aplicación.
`/res/raw/`	Contiene recursos adicionales, normalmente en formato distinto a XML, que no se incluyan en el resto de carpetas de recursos.

Como ejemplo, para un proyecto nuevo Android, se crean los siguientes recursos para la aplicación:



Carpeta /gen/

Contiene una serie de elementos de código generados automáticamente al compilar el proyecto. Cada vez que generamos nuestro proyecto, la maquinaria de compilación de Android genera por nosotros una serie de ficheros fuente en java dirigidos al control de los recursos de la aplicación.



El más importante es el que se puede observar en la imagen, el fichero `R.java`, y la clase `R`.

Esta clase `R` contendrá en todo momento una serie de constantes con los ID de todos los recursos de la aplicación incluidos en la carpeta `/res/`, de forma que podamos acceder fácilmente a estos recursos desde nuestro código a través de este dato. Así, por ejemplo, la constante `R.drawable.icon` contendrá el ID de la imagen "*icon.png*" contenida en la carpeta `/res/drawable/`. Veamos como ejemplo la clase R creada por defecto para un proyecto nuevo:

```
package net.sgoliver;
 public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
```

```
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

Carpeta /assets/

Contiene todos los demás ficheros auxiliares necesarios para la aplicación (y que se incluirán en su propio paquete), como por ejemplo ficheros de configuración, de datos, etc.

La diferencia entre los recursos incluidos en la carpeta `/res/raw/` y los incluidos en la carpeta `/assets/` es que para los primeros se generará un ID en la clase `R` y se deberá acceder a ellos con los diferentes métodos de acceso a recursos. Para los segundos sin embargo no se generarán ID y se podrá acceder a ellos por su ruta como a cualquier otro fichero del sistema. Usaremos uno u otro según las necesidades de nuestra aplicación.

Fichero AndroidManifest.xml

Contiene la definición en XML de los aspectos principales de la aplicación, como por ejemplo su identificación (nombre, versión, icono, …), sus componentes (pantallas, mensajes, …), o los permisos necesarios para su ejecución. Veremos más adelante más detalles de este fichero.

En el siguiente apartado veremos los componentes software principales con los que podemos construir una aplicación Android.

Componentes de una aplicación Android

En el apartado anterior vimos la estructura de un proyecto Android y aprendimos dónde colocar cada uno de los elementos que componen una aplicación, tanto elementos de software como recursos gráficos o de datos. En éste nuevo post vamos a centrarnos específicamente en los primeros, es decir, veremos los distintos tipos de componentes de software con los que podremos construir una aplicación Android.

En Java o .NET estamos acostumbrados a manejar conceptos como ventana, control, eventos o servicios como los elementos básicos en la construcción de una aplicación.

Pues bien, en Android vamos a disponer de esos mismos elementos básicos aunque con un pequeño cambio en la terminología y el enfoque. Repasemos los componentes principales que pueden formar parte de una aplicación Android [Por claridad, y para evitar confusiones al consultar documentación en inglés, intentaré traducir lo menos posible los nombres originales de los componentes].

Activity

Las actividades (*activities*) representan el componente principal de la interfaz gráfica de una aplicación Android. Se puede pensar en una actividad como el elemento análogo a una ventana en cualquier otro lenguaje visual.

View

Los objetos *view* son los componentes básicos con los que se construye la interfaz gráfica de la aplicación, análogo por ejemplo a los *controles* de Java o .NET. De inicio, Android pone a nuestra disposición una gran cantidad de controles básicos, como cuadros de texto, botones, listas desplegables o imágenes, aunque también existe la posibilidad de extender la funcionalidad de estos controles básicos o crear nuestros propios controles personalizados.

Service

Los servicios son componentes sin interfaz gráfica que se ejecutan en segundo plano. En concepto, son exactamente iguales a los servicios presentes en cualquier otro sistema operativo. Los servicios pueden realizar cualquier tipo de acciones, por ejemplo actualizar datos, lanzar notificaciones, o incluso mostrar elementos visuales (p.ej. activities) si se necesita en algún momento la interacción con del usuario.

Content Provider

Un *content provider* es el mecanismo que se ha definido en Android para compartir datos entre aplicaciones. Mediante estos componentes es posible compartir determinados datos de nuestra aplicación sin mostrar detalles sobre su almacenamiento interno, su estructura, o su implementación. De la misma forma, nuestra aplicación podrá acceder a los datos de otra a través de los *content provider* que se hayan definido.

Broadcast Receiver

Un *broadcast receiver* es un componente destinado a detectar y reaccionar ante determinados mensajes o eventos globales generados por el sistema (por ejemplo: "Batería baja", "SMS recibido", "Tarjeta SD insertada", …) o por otras aplicaciones (cualquier aplicación puede generar mensajes (*intents*, en terminología Android) broadcast, es decir, no dirigidos a una aplicación concreta sino a cualquiera que quiera escucharlo).

Widget

Los *widgets* son elementos visuales, normalmente interactivos, que pueden mostrarse en la pantalla principal (*home screen*) del dispositivo Android y recibir actualizaciones periódicas. Permiten mostrar información de la aplicación al usuario directamente sobre la pantalla principal.

Intent

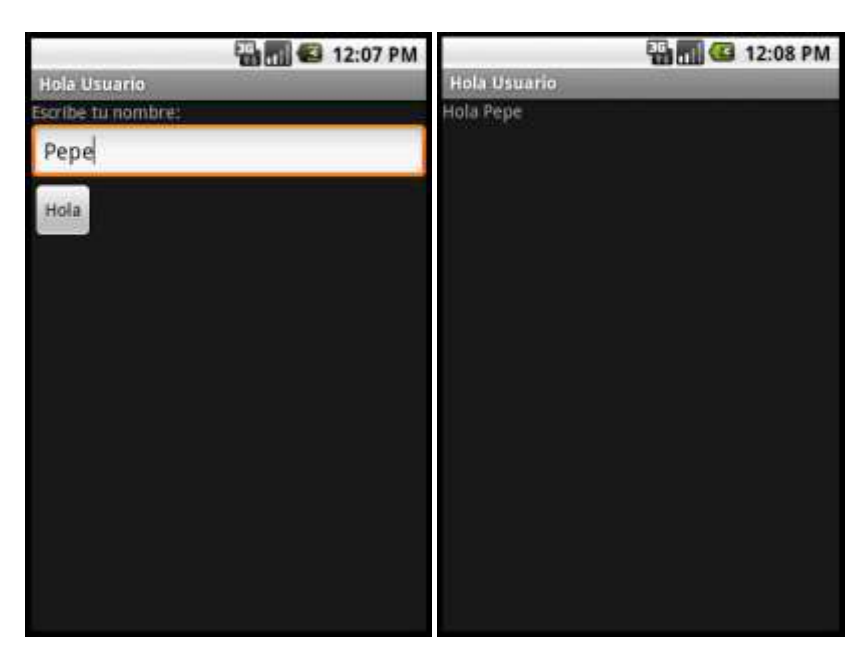
Un *intent* es el elemento básico de comunicación entre los distintos componentes Android que hemos descrito anteriormente. Se pueden entender como los mensajes o peticiones que son enviados entre los distintos componentes de una aplicación o entre distintas aplicaciones. Mediante un intent se puede mostrar una actividad desde cualquier otra, iniciar un servicio, enviar un mensaje broadcast, iniciar otra aplicación, etc.

En el siguiente apartado empezaremos ya a ver algo de código, analizando al detalle una aplicación sencilla.

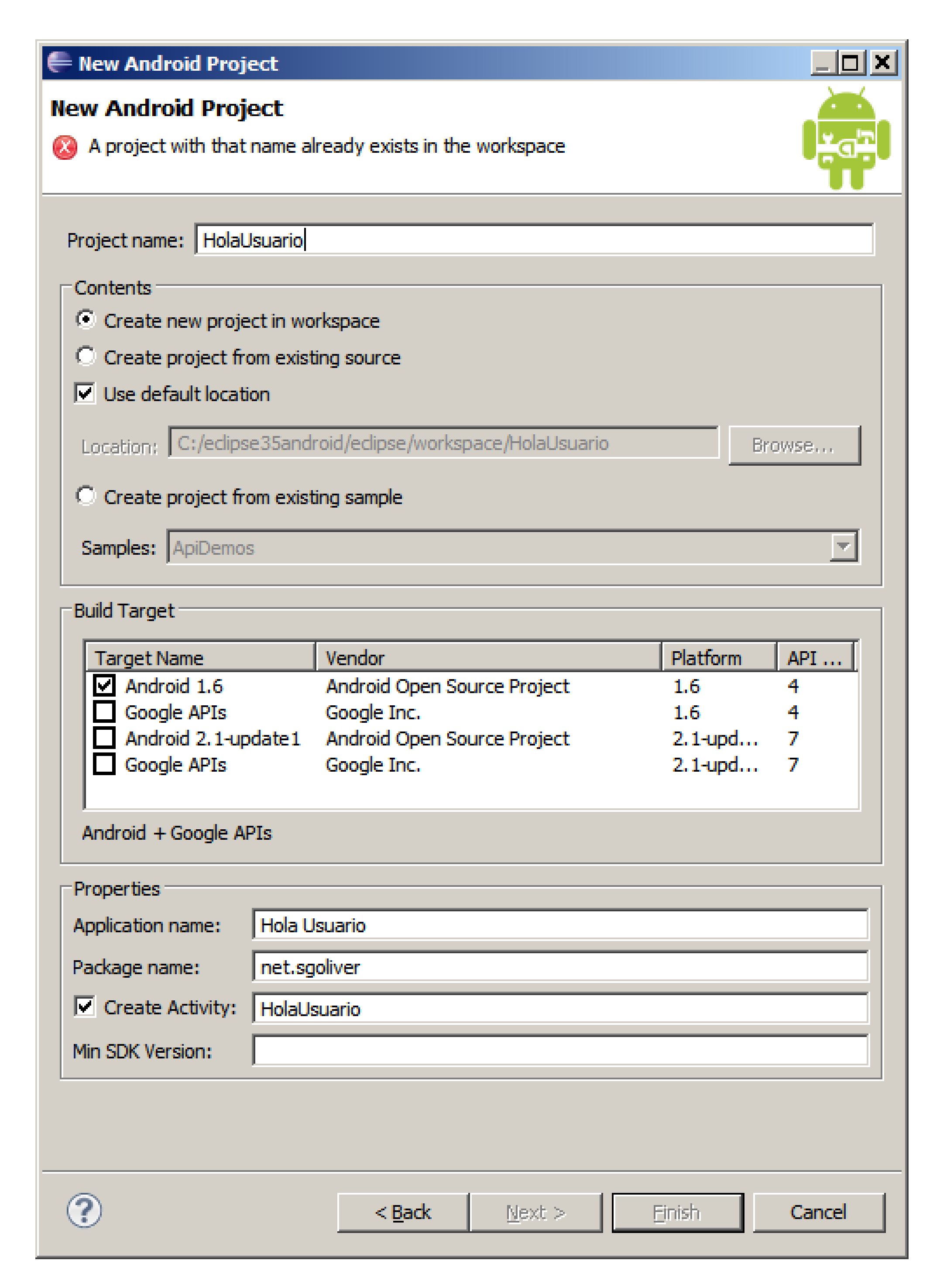
Desarrollando una aplicación Android sencilla

Después de instalar nuestro entorno de desarrollo para Android y comentar la estructura básica de un proyecto y los diferentes componentes software que podemos utilizar ya es hora de empezar a escribir algo de código. Y como siempre lo mejor es empezar por escribir una aplicación sencilla.

En un principio me planteé analizar en este post el clásico *Hola Mundo* pero más tarde me pareció que se iban a quedar algunas cosas básicas en el tintero. Así que he versionado a mi manera el *Hola Mundo* transformándolo en algo así como un *Hola Usuario*, que es igual de sencilla pero añade un par de cosas interesantes de contar. La aplicación constará de dos pantallas, por un lado la pantalla principal que solicitará un nombre al usuario y una segunda pantalla en la que se mostrará un mensaje personalizado para el usuario. Sencillo, inútil, pero aprenderemos muchos conceptos básicos, que para empezar no está mal.



En primer lugar vamos a crear un nuevo proyecto Android tal como vimos al final del primer post de la serie. Llamaremos al proyecto "*HolaUsuario*", indicaremos como target por ejemplo "Android 1.6", daremos un nombre a la aplicación e indicaremos que se cree una actividad llamada `HolaUsuario`.



Como ya vimos esto nos crea la estructura de carpetas del proyecto y todos los ficheros necesarios de un *Hola Mundo* básico, es decir, una sola pantalla donde se muestra únicamente un mensaje fijo.

Lo primero que vamos a hacer es diseñar nuestra pantalla principal modificando la que Eclipse nos ha creado por defecto. ¿Pero dónde y cómo se define cada pantalla de la aplicación? En Android, el diseño y la lógica de una pantalla están separados en dos ficheros distintos. Por un lado, en el fichero `/res/layout/main.xml` tendremos el diseño puramente visual de la pantalla definido como fichero XML y por otro lado, en el fichero `/src/paquetejava/HolaUsuario.java`, encontraremos el código java que determina la lógica de la pantalla.

Vamos a modificar en primer lugar el aspecto de la ventana principal de la aplicación añadiendo los controles (*views*) que vemos en la primera captura de pantalla. Para ello, vamos a sustituir el contenido del fichero `main.xml` por el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >

    <TextView android:text="@string/nombre"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />

    <EditText android:id="@+id/TxtNombre"
        android:layout_height="wrap_content"
        android:layout_width="fill_parent" />

    <Button android:id="@+id/BtnHola"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hola" />

</LinearLayout>
```

En este XML se definen los elementos visuales que componen la interfaz de nuestra pantalla principal y se especifican todas sus propiedades. No nos detendremos mucho por ahora en cada detalle, pero expliquemos un poco lo que vemos en el fichero.

Lo primero que nos encontramos es un elemento `LinearLayout`. Los *layout* son elementos no visibles que determinan cómo se van a distribuir en el espacio los controles que incluyamos en su interior. Los programadores java, y más concretamente de *Swing*, conocerán este concepto perfectamente. En este caso, un `LinearLayout` distribuirá los controles uno tras otro y en la orientación que indique su propiedad `android:orientation`.

Dentro del *layout* hemos incluido 3 controles: una etiqueta (`TextView`), un cuadro de texto (`EditText`), y un botón (`Button`). En todos ellos hemos establecido las siguientes propiedades:

- `android:id`. ID del control, con el que podremos identificarlo más tarde en nuestro código. Vemos que el identificador lo escribimos precedido de "`@+id/`". Esto tendrá como efecto que al compilarse el proyecto se genere automáticamente una nueva constante en la clase `R` para dicho control.
- `android:text`. Texto del control. En Android, el texto de un control se puede especificar directamente, o bien utilizar alguna de las cadenas de texto definidas en los recursos del proyecto (fichero `strings.xml`), en cuyo caso indicaremos su identificador precedido del prefijo "`@string/`".
- `android:layout_height` y `android:layout_width`. Dimensiones del control con respecto al layout que lo contiene. Esta propiedad tomará normalmente los valores "`wrap_content`" para indicar que las dimensiones del control se ajustarán al contenido del mismo, o bien "`fill_parent`" para indicar que el ancho o el alto del control se ajustará al ancho o alto del layout contenedor respectivamente.

Con esto ya tenemos definida la presentación visual de nuestra ventana principal de la aplicación. De igual forma definiremos la interfaz de la segunda pantalla, creando un nuevo fichero llamado `frmmensaje.xml`, y añadiendo esta vez tan solo una etiqueta (`TextView`) para mostrar el mensaje personalizado al usuario. Veamos cómo quedaría nuestra segunda pantalla:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content">

<TextView android:id="@+id/TxtMensaje"
    android:layout_height="wrap_content"
    android:layout_width="fill_parent"
    android:text="$mensaje"></TextView>

</LinearLayout>
```

Una vez definida la interfaz de las pantallas de la aplicación deberemos implementar la lógica de la misma. Como ya hemos comentado, la lógica de la aplicación se definirá en ficheros java independientes. Para la pantalla principal ya tenemos creado un fichero por defecto llamado `HolaUsuario.java`. Empecemos por comentar su código por defecto:

```
public class HolaUsuario extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Como ya vimos en un apartado anterior, las diferentes pantallas de una aplicación Android se definen mediante objetos de tipo `Activity`. Por tanto, lo primero que encontramos en nuestro fichero java es la definición de una nueva clase `HolaUsuario` que extiende a `Activity`. El único método que sobrescribiremos de esta clase será el método `OnCreate`, llamado cuando se crea por primera vez la actividad. En este método lo único que encontramos en principio, además de la llamada a su implementación en la clase padre, es la llamada al método `setContentView(R.layout.main)`. Con esta llamada estaremos indicando a Android que debe establecer como interfaz gráfica de esta actividad la definida en el recurso `R.layout.main`, que no es más que la que hemos especificado en el fichero `/res/layout/main.xml`. Una vez más vemos la utilidad de las diferentes constantes de recursos creadas automáticamente en la clase R al compilar el proyecto.

En principio vamos a crear una nueva actividad para la segunda pantalla de la aplicación análoga a ésta primera, para lo que crearemos una nueva clase `FrmMensaje` que extienda de `Activity` y que implemente el método `onCreate` indicando que utilice la interfaz definida en `R.layout.frmmensaje`.

```
public class FrmMensaje extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.frmmensaje);
    }
}
```

Como vemos, el código incluido por defecto en estas clases lo único que hace es generar la interfaz de la actividad. A partir de aquí nosotros tendremos que incluir el resto de la lógica de la aplicación. Y vamos a empezar con la actividad principal `HolaUsuario`, obteniendo una referencia a los diferentes controles de la interfaz que necesitemos manipular, en nuestro caso sólo el cuadro de texto y el botón. Para ello utilizaremos el método `findViewById()` indicando el ID de cada control, definidos como siempre en la clase R:

```
final EditText txtNombre = (EditText)findViewById(R.id.TxtNombre);
final Button btnHola = (Button)findViewById(R.id.BtnHola);
```

Una vez tenemos acceso a los diferentes controles, ya sólo nos queda implementar las acciones a tomar cuando pulsemos el botón de la pantalla. Para ello implementaremos el evento `onClick` de dicho botón, veamos cómo:

```
btnHola.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        Intent intent = new Intent(HolaUsuario.this, FrmMensaje.class);

        Bundle bundle = new Bundle();
        bundle.putString("NOMBRE", txtNombre.getText().toString());
        intent.putExtras(bundle);

        startActivity(intent);
    }
});
```

Como ya indicamos en el apartado anterior, la comunicación entre los distintos componentes y aplicaciones en Android se realiza mediante *intents*, por lo que el primer paso será crear un objeto de este tipo. Existen varias variantes del constructor de la clase `Intent`, cada una de ellas dirigida a unas determinadas acciones, pero en nuestro caso particular vamos a utilizar el intent para llamar a una actividad desde otra de la misma aplicación, para lo que pasaremos al constructor una referencia a la propia actividad llamadora (`HolaUsuario.this`), y la clase de la actividad llamada (`FrmMensaje.class`).

Si quisiéramos tan sólo mostrar una nueva actividad ya tan sólo nos quedaría llamar a `startActivity()` pasándole como parámetro el intent creado. Pero en nuestro ejemplo queremos también pasarle cierta información a la actividad, concretamente el nombre que introduzca el usuario en el cuadro de texto. Para hacer esto vamos a crear un objeto `Bundle`, que puede contener una lista de pares clave-valor con toda la información a pasar entre las actividades. En nuestro caso sólo añadiremos un dato de tipo `String` mediante el método `putString(clave, valor)`. Tras esto añadiremos la información al intent mediante el método `putExtras(bundle)`.

Finalizada la actividad principal de la aplicación pasamos ya a la secundaria. Comenzaremos de forma análoga a la anterior, ampliando el método `onCreate` obteniendo las referencias a los objetos que manipularemos, esta vez sólo la etiqueta de texto. Tras esto viene lo más interesante, debemos recuperar la información pasada desde la actividad principal y asignarla como texto de la etiqueta. Para ello accederemos en primer lugar al intent que ha originado la actividad actual mediante el método `getIntent()` y recuperaremos su información asociada (objeto `Bundle`) mediante el método `getExtras()`.

Hecho esto tan sólo nos queda construir el texto de la etiqueta mediante su método `setText(texto)` y recuperando el valor de nuestra clave almacenada en el objeto `Bundle` mediante `getString(clave)`.

```
public class FrmMensaje extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.frmmensaje);

        TextView txtMensaje = (TextView)findViewById(R.id.TxtMensaje);

        Bundle bundle = getIntent().getExtras();

        txtMensaje.setText("Hola " + bundle.getString("NOMBRE"));
    }
}
```

Con esto hemos concluido la lógica de las dos pantallas de nuestra aplicación y tan sólo nos queda un paso importante para finalizar nuestro desarrollo. Como ya indicamos en alguna ocasión, toda aplicación Android utiliza un fichero especial en formato XML (`AndroidManifest.xml`) para definir, entre otras cosas, los diferentes elementos que la componen. Por tanto, todas las actividades de nuestra aplicación deben quedar convenientemente recogidas en este fichero. La actividad principal ya debe aparecer puesto que se creó de forma automática al crear el nuevo proyecto Android, por lo que debemos añadir tan sólo la segunda. Para este ejemplo nos limitaremos a incluir la actividad en el XML, más adelante veremos que opciones adicionales podemos especificar.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="net.sgoliver" android:versionCode="1"
      android:versionName="1.0">

    <application android:icon="@drawable/icon"
                 android:label="@string/app_name">
        <activity android:name=".HolaUsuario"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity android:name=".FrmMensaje"></activity>
     </application>
    <uses-sdk android:minSdkVersion="4" />
</manifest>
```

Una vez llegado aquí, si todo ha ido bien, deberíamos poder ejecutar el proyecto sin errores y probar nuestra aplicación en el emulador.

Espero que esta aplicación de ejemplo os sea de ayuda para aprender temas básicos en el desarrollo para Android, como por ejemplo la definición de la interfaz gráfica, el código java necesario para acceder y manipular los elementos de dicha interfaz, o la forma de comunicar diferentes actividades de Android. En los apartados siguientes veremos algunos de estos temas de forma mucho más específica.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-HolaUsuario.zip

2

Interfaz de Usuario

II. Interfaz de Usuario

Layouts

En el apartado anterior, donde desarrollamos una sencilla aplicación Android desde cero, ya hicimos algunos comentarios sobre los *layouts*. Como ya indicamos, los *layouts* son elementos no visuales destinados a controlar la distribución, posición y dimensiones de los controles que se insertan en su interior. Estos componentes extienden a la clase base `ViewGroup`, como muchos otros componentes contenedores, es decir, capaces de contener a otros controles. En el post anterior conocimos la existencia de un tipo concreto de layout, `LinearLayout`, aunque Android nos proporciona algunos otros. Veamos cuántos y cuáles.

FrameLayout

Éste es el más simple de todos los layouts de Android. Un `FrameLayout` coloca todos sus controles hijos alineados con su esquina superior izquierda, de forma que cada control quedará oculto por el control siguiente (a menos que éste último tenga transparencia). Por ello, suele utilizarse para mostrar un único control en su interior, a modo de contenedor (*placeholder*) sencillo para un sólo elemento sustituible, por ejemplo una imagen.

Los componentes incluidos en un `FrameLayout` podrán establecer sus propiedades `android:layout_width` y `android:layout_height`, que podrán tomar los valores "`fill_parent`" (para que el control hijo tome la dimensión de su layout contenedor) o "`wrap_content`" (para que el control hijo tome la dimensión de su contenido).

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <EditText android:id="@+id/TxtNombre"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />

</FrameLayout>
```

LinearLayout

El siguiente layout Android en cuanto a nivel de complejidad es el `LinearLayout`. Este layout apila uno tras otro todos sus elementos hijos de forma horizontal o vertical según se establezca su propiedad `android:orientation`.

Al igual que en un `FrameLayout`, los elementos contenidos en un `LinearLayout` pueden establecer sus propiedades `android:layout_width` y `android:layout_height` para determinar sus dimensiones dentro del layout. Pero en el caso de un `LinearLayout`, tendremos otro parámetro con el que jugar, la propiedad `android:layout_weight`.

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <EditText android:id="@+id/TxtNombre"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />

    <Button android:id="@+id/BtnAceptar"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent" />

</LinearLayout>
```

Esta propiedad nos va a permitir dar a los elementos contenidos en el layout unas dimensiones proporcionales entre ellas. Esto es más difícil de explicar que de comprender con un ejemplo. Si incluimos en un `LinearLayout` vertical dos cuadros de texto (`EditText`) y a uno de ellos le establecemos un `layout_weight="1"` y al otro un `layout_weight="2"` conseguiremos como efecto que toda la superficie del layout quede ocupada por los dos cuadros de texto y que además el segundo sea el doble (relación entre sus propiedades `weight`) de alto que el primero.

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <EditText android:id="@+id/TxtDato1"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layout_weight="1" />

    <EditText android:id="@+id/TxtDato2"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layout_weight="2" />

</LinearLayout>
```

Así pues, a pesar de la simplicidad aparente de este layout resulta ser lo suficiente versátil como para sernos de utilidad en muchas ocasiones.

TableLayout

Un `TableLayout` permite distribuir sus elementos hijos de forma tabular, definiendo las filas y columnas necesarias, y la posición de cada componente dentro de la tabla.

La estructura de la tabla se define de forma similar a como se hace en HTML, es decir, indicando las filas que compondrán la tabla (objetos `TableRow`), y dentro de cada fila las columnas necesarias, con la salvedad de que no existe ningún objeto especial para definir una columna (algo así como un `TableColumn`) sino que directamente insertaremos los controles necesarios dentro del `TableRow` y cada componente insertado (que puede ser un control sencillo o incluso otro `ViewGroup`) corresponderá a una columna de la tabla. De esta forma,

el número final de filas de la tabla se corresponderá con el número de elementos `TableRow` insertados, y el número total de columnas quedará determinado por el número de componentes de la fila que más componentes contenga.

Por norma general, el ancho de cada columna se corresponderá con el ancho del mayor componente de dicha columna, pero existen una serie de propiedades que nos ayudarán a modificar este comportamiento:

- `android:stretchColumns`. Indicará las columnas que pueden expandir para absorver el espacio libre dejado por las demás columnas a la derecha de la pantalla.
- `android:shrinkColumns`. Indicará las columnas que se pueden contraer para dejar espacio al resto de columnas que se puedan salir por la derecha de la pantalla.
- `android:collapseColumns`. Indicará las columnas de la tabla que se quieren ocultar completamente.

Todas estas propiedades del `TableLayout` pueden recibir una lista de índices de columnas separados por comas (ejemplo: `android:stretchColumns="1,2,3"`) o un asterisco para indicar que debe aplicar a todas las columnas (ejemplo: `android:stretchColumns="*"`).

Otra característica importante es la posibilidad de que una celda determinada pueda ocupar el espacio de varias columnas de la tabla (análogo al atributo `colspan` de HTML). Esto se indicará mediante la propiedad `android:layout_span` del componente concreto que deberá tomar dicho espacio.

Veamos un ejemplo con varios de estos elementos:

```
<TableLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1" >

    <TableRow>
        <TextView android:text="Celda 1.1" />
        <TextView android:text="Celda 1.2" />
    </TableRow>

    <TableRow>
        <TextView android:text="Celda 2.1" />
        <TextView android:text="Celda 2.2" />
    </TableRow>

    <TableRow>
        <TextView android:text="Celda 3"
               android:layout_span="2" />
    </TableRow>
</TableLayout>
```

RelativeLayout

El último tipo de layout que vamos a ver es el `RelativeLayout`. Este layout permite especificar la posición de cada elemento de forma relativa a su elemento padre o a cualquier otro elemento incluido en el propio layout. De esta forma, al incluir un nuevo elemento A

podremos indicar por ejemplo que debe colocarse debajo del elemento B y alineado a la derecha del layout padre. Veamos esto en el ejemplo siguiente:

```
<RelativeLayout
        xmlns:android="http://schemas.android.com/apk/res/android
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >

        <EditText android:id="@+id/TxtNombre"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content" />

        <Button android:id="@+id/BtnAceptar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_below="@id/TxtNombre"
            android:layout_alignParentRight="true" />
</RelativeLayout>
```

En el ejemplo, el botón `BtnAceptar` se colocará debajo del cuadro de texto `TxtNombre` (`android:layout_below="@id/TxtNombre"`) y alineado a la derecha del layout padre (`android:layout_alignParentRight="true"`), además de dejar un margen a su izquierda de 10 pixeles (`android:layout_marginLeft="10px"`).

Al igual que estas tres propiedades, en un `RelativeLayout` tendremos un sinfín de propiedades para colocar cada control justo donde queramos. Veamos las principales [creo que sus propios nombres explican perfectamente la función de cada una]:

Tipo	**Propiedades**
Posición relativa a otro control	android:layout_above android:layout_below android:layout_toLeftOf android:layout_toRightOf android:layout_alignLeft android:layout_alignRight android:layout_alignTop android:layout_alignBottom android:layout_alignBaseline
Posición relativa al layout padre	android:layout_alignParentLeft android:layout_alignParentRight android:layout_alignParentTop android:layout_alignParentBottom android:layout_centerHorizontal android:layout_centerVertical android:layout_centerInParent
Opciones de margen (también disponibles para el resto de layouts)	android:layout_margin android:layout_marginBottom android:layout_marginTop android:layout_marginLeft android:layout_marginRight
Opciones de espaciado o *padding* (también disponibles para el resto	android:padding android:paddingBottom android:paddingTop

de layouts)	android:paddingLeft android:paddingRight

En próximos apartados veremos otros elementos comunes que extienden a ViewGroup, como por ejemplo las vistas de tipo lista (ListView), de tipo grid (GridView), y en pestañas (TabHost/TabWidget).

Botones

En el apartado anterior hemos visto los distintos tipos de *layout* con los que contamos en Android para distribuir los controles de la interfaz por la pantalla del dispositivo. En los próximos apartados vamos a hacer un repaso de los diferentes controles que pone a nuestra disposición la plataforma de desarrollo de este sistema operativo. Empezaremos con los controles más básicos y seguiremos posteriormente con algunos algo más elaborados.

En esta primera parada sobre el tema nos vamos a centrar en los diferentes tipos de botones y cómo podemos personalizarlos. El SDK de Android nos proporciona tres tipos de botones: el clásico (Button), el de tipo *on/off* (ToggleButton), y el que puede contener una imagen (ImageButton). En la imagen siguiente vemos el aspecto por defecto de estos tres controles.



No vamos a comentar mucho sobre ellos dado que son controles de sobra conocidos por todos, ni vamos a enumerar todas sus propiedades porque existen decenas. A modo de referencia, a medida que los vayamos comentando iré poniendo enlaces a su página de la documentación oficial de Android para poder consultar todas sus propiedades en caso de necesidad.

Control Button [API]

Un control de tipo Button es el botón más básico que podemos utilizar. En el ejemplo siguiente definimos un botón con el texto "*Púlsame*" asignando su propiedad android:text. Además de esta propiedad podríamos utilizar muchas otras como el color de fondo (android:background), estilo de fuente (android:typeface), color de fuente (android:textcolor), tamaño de fuente (android:textSize), etc.

```
<Button android:id="@+id/BtnBoton1"
    android:text="Púlsame"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Control ToggleButton [API]

Un control de tipo `ToggleButton` es un tipo de botón que puede permanecer en dos estados, pulsado/no_pulsado. En este caso, en vez de definir un sólo texto para el control definiremos dos, dependiendo de su estado. Así, podremos asignar las propiedades `android:textOn` y `android:textoOff` para definir ambos textos.

Veamos un ejemplo a continuación.

```
<ToggleButton android:id="@+id/BtnBoton2"
    android:textOn="ON"
    android:textOff="OFF"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Control ImageButton [API]

En un control de tipo `ImageButton` podremos definir una imagen a mostrar en vez de un texto, para lo que deberemos asignar la propiedad `android:src`. Normalmente asignaremos esta propiedad con el descriptor de algún recurso que hayamos incluido en la carpeta `/res/drawable`. Así, por ejemplo, en nuestro caso hemos incluido una imagen llamada "*ok.png*" por lo que haremos referencia al recurso "`@drawable/ok`".

```
<ImageButton android:id="@+id/BtnBoton3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/ok" />
```

Eventos de un botón

Como podéis imaginar, aunque estos controles pueden lanzar muchos otros eventos, el más común de todos ellos y el que querremos capturar en la mayoría de las ocasiones es el evento `onClick`. Para definir la lógica de este evento tendremos que implementarla definiendo un nuevo objeto `View.OnClickListener()` y asociándolo al botón mediante el método `setOnClickListener()`. La forma más habitual de hacer esto es la siguiente:

```
final Button btnBoton1 = (Button)findViewById(R.id.BtnBoton1);

btnBoton1.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View arg0)
    {
        lblMensaje.setText("Botón 1 pulsado!");
    }
});
```

En el caso de un botón de tipo `ToggleButton` suele ser de utilidad conocer en qué estado ha quedado el botón tras ser pulsado, para lo que podemos utilizar su método `isChecked()`. En el siguiente ejemplo se comprueba el estado del botón tras ser pulsado y se realizan acciones distintas según el resultado.

```
final ToggleButton btnBoton2 = (ToggleButton)findViewById(R.id.BtnBoton2);

btnBoton2.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View arg0)
    {
        if(btnBoton2.isChecked())
            lblMensaje.setText("Botón 2: ON");
        else
            lblMensaje.setText("Botón 2: OFF");
    }
});
```

Personalizar el aspecto un botón [y otros controles]

En la imagen anterior vimos el aspecto que presentan por defecto los tres tipos de botones disponibles. Pero, ¿y si quisiéramos personalizar su aspecto más allá de cambiar un poco el tipo o el color de la letra o el fondo?

Para cambiar la forma de un botón podríamos simplemente asignar una imagen a la propiedad `android:background`, pero esta solución no nos serviría de mucho porque siempre se mostraría la misma imagen incluso con el botón pulsado, dando poca sensación de elemento "clickable".

La solución perfecta pasaría por tanto por definir diferentes imágenes de fondo dependiendo del estado del botón. Pues bien, Android nos da total libertad para hacer esto mediante el uso de selectores. Un selector se define mediante un fichero XML localizado en la carpeta `/res/drawable`, y en él se pueden establecer los diferentes valores de una propiedad determinada de un control dependiendo de su estado.

Por ejemplo, si quisiéramos dar un aspecto plano a un botón `ToggleButton`, podríamos diseñar las imágenes necesarias para los estados "pulsado" (en el ejemplo `toggle_on.png`) y "no pulsado" (en el ejemplo `toggle_off.png`) y crear un selector como el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<selector
    xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:state_checked="false"
            android:drawable="@drawable/toggle_off" />
    <item android:state_checked="true"
            android:drawable="@drawable/toggle_on" />

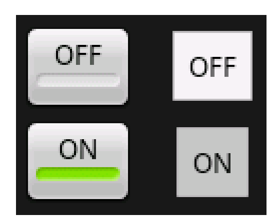
</selector>
```

Este selector lo guardamos por ejemplo en un fichero llamado `toggle_style.xml` y lo colocamos como un recurso más en nuestra carpeta de recursos `/res/drawable`.

Hecho esto, tan sólo bastaría hacer referencia a este nuevo recurso que hemos creado en la propiedad `android:background` del botón:

```
<ToggleButton android:id="@+id/BtnBoton4"
    android:textOn="ON"
    android:textOff="OFF"
    android:padding="10dip"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="@drawable/toggle_style"/>
```

En la siguiente imagen vemos el aspecto por defecto de un ToggleButton y cómo ha quedado nuestro ToggleButton personalizado.



El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-botones.zip

Imágenes, etiquetas y cuadros de texto

En este apartado nos vamos a centrar en otros tres componentes básicos imprescindibles en nuestras aplicaciones: las imágenes (ImageView), las etiquetas (TextView) y por último los cuadros de texto (EditText).

Control ImageView [API]

El control ImageView permite mostrar imágenes en la aplicación. La propiedad más interesante es android:src, que permite indicar la imagen a mostrar. Nuevamente, lo normal será indicar como origen de la imagen el identificador de un recurso de nuestra carpeta /res/drawable, por ejemplo android:src="@drawable/unaimagen". Además de esta propiedad, existen algunas otras útiles en algunas ocasiones como las destinadas a establecer el tamaño máximo que puede ocupar la imagen, android:maxWidth y android:maxHeight.

```
<ImageView android:id="@+id/ImgFoto"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:src="@drawable/icon" />
```

En la lógica de la aplicación, podríamos establecer la imagen mediante el método setImageResorce(…), pasándole el ID del recurso a utilizar como contenido de la imagen.

```
ImageView img = (ImageView)findViewById(R.id.ImgFoto);
img.setImageResource(R.drawable.icon);
```

Control TextView [API]

El control TextView es otro de los clásicos en la programación de GUIs, las etiquetas de texto, y se utiliza para mostrar un determinado texto al usuario. Al igual que en el caso de los botones, el texto del control se establece mediante la propiedad android:text. A parte de esta propiedad, la naturaleza del control hace que las más interesantes sean las que establecen el formato del texto mostrado, que al igual que en el caso de los botones son las siguientes: android:background (color de fondo), android:textColor (color del texto), android:textSize (tamaño de la fuente) y android:typeface (estilo del texto: negrita, cursiva, …).

```
<TextView android:id="@+id/LblEtiqueta"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Escribe algo:"
android:background="#AA44FF"
android:typeface="monospace" />
```

De igual forma, también podemos manipular estas propiedades desde nuestro código. Como ejemplo, en el siguiente fragmento recuperamos el texto de una etiqueta con getText(), y posteriormente le concatenamos unos números, actualizamos su contenido mediante setText() y le cambiamos su color de fondo con setBackgroundColor().

```
final TextView lblEtiqueta = (TextView)findViewById(R.id.LblEtiqueta);
String texto = lblEtiqueta.getText().toString();
texto += "123";
lblEtiqueta.setText(texto);
```

Control EditText [API]

El control EditText es el componente de edición de texto que proporciona la plataforma Android. Permite la introducción y edición de texto por parte del usuario, por lo que en tiempo de diseño la propiedad más interesante a establecer, además de su posición/tamaño y formato, es el texto a mostrar, atributo android:text.

```
<EditText android:id="@+id/TxtTexto"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:layout_below="@id/LblEtiqueta" />
```

De igual forma, desde nuestro código podremos recuperar y establecer este texto mediante los métodos getText() y setText(nuevoTexto) respectivamente:

```
final EditText txtTexto = (EditText)findViewById(R.id.TxtTexto);
String texto = txtTexto.getText().toString();
txtTexto.setText("Hola mundo!");
```

Un detalle que puede haber pasado desapercibido. ¿Os habéis fijado en que hemos tenido que hacer un toString() sobre el resultado de getText()? La explicación para esto es que el método getText() no devuelve un String sino un objeto de tipo Editable, que a su vez implementa la interfaz Spannable. Y esto nos lleva a la característica más interesante

del control EditText, y es que no sólo nos permite editar texto plano sino también texto enriquecido o con formato. Veamos cómo y qué opciones tenemos, y para empezar comentemos algunas cosas sobre los objetos Spannable.

Interfaz Spanned

Un objeto de tipo Spanned es algo así como una cadena de caracteres (deriva de la interfaz CharSequence) en la que podemos insertar otros objetos a modo de marcas o etiquetas (*spans*) asociados a rangos de caracteres. De esta interfaz deriva la interfaz Spannable, que permite la modificación de estas marcas, y a su vez de ésta última deriva la interfaz Editable, que permite además la modificación del texto.

Aunque en el apartado en el que nos encontramos nos interesaremos principalmente por las marcas de formato de texto, en principio podríamos insertar cualquier tipo de objeto.

Existen muchos tipos de *spans* predefinidos en la plataforma que podemos utilizar para dar formato al texto, entre ellos:

- TypefaceSpan. Modifica el tipo de fuente.
- StyleSpan. Modifica el estilo del texto (negrita, cursiva, …).
- ForegroudColorSpan. Modifica el color del texto.
- AbsoluteSizeSpan. Modifica el tamaño de fuente.

De esta forma, para crear un nuevo objeto Editable e insertar una marca de formato podríamos hacer lo siguiente:

```
//Creamos un nuevo objeto de tipo Editable
Editable str = Editable.Factory.getInstance().newEditable("Esto es un
simulacro.");

//Marcamos cono fuente negrita la palabra "simulacro"
str.setSpan(new StyleSpan(android.graphics.Typeface.BOLD), 11, 19,
Spannable.SPAN_EXCLUSIVE_EXCLUSIVE);
```

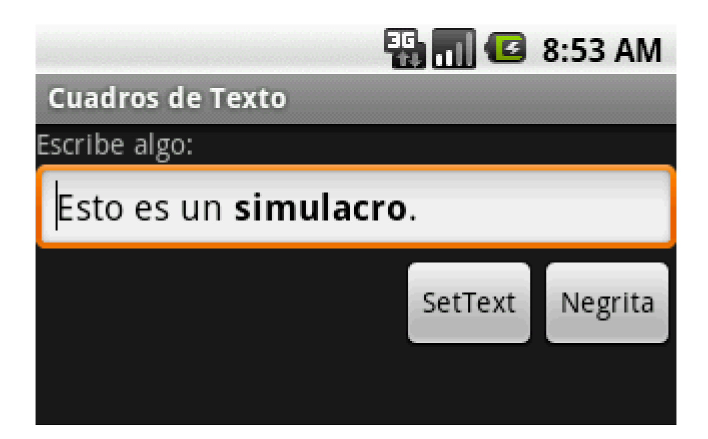
En este ejemplo estamos insertando un span de tipo StyleSpan para marcar un fragmento de texto con estilo negrita. Para insertarlo utilizamos el método setSpan(), que recibe como parámetro el objeto Span a insertar, la posición inicial y final del texto a marcar, y un flag que indica la forma en la que el span se podrá extender al insertarse nuevo texto.

Texto con formato en controles TextView y EditText

Hemos visto cómo crear un objeto Editable y añadir marcas de formato al texto que contiene, pero todo esto no tendría ningún sentido si no pudiéramos visualizarlo. Como ya podéis imaginar, los controles TextView y EditText nos van a permitir hacer esto. Vemos qué ocurre si asignamos al nuestro control EditText el objeto Editable que hemos creado antes:

```
txtTexto.setText(str);
```

Tras ejecutar este código veremos cómo efectivamente en el cuadro de texto aparece el mensaje con el formato esperado:



Ya hemos visto cómo asignar texto con y sin formato a un cuadro de texto, pero ¿qué ocurre a la hora de recuperar texto con formato desde el control?. Ya vimos que la función `getText()` devuelve un objeto de tipo `Editable` y que sobre éste podíamos hacer un `toString()`. Pero con esta solución estamos perdiendo todo el formato del texto, por lo que no podríamos por ejemplo salvarlo a una base de datos.

La solución a esto último pasa obviamente por recuperar directamente el objeto Editable y serializarlo de algún modo, mejor aún si es en un formato estándar. Pues bien, en Android este trabajo ya nos viene hecho de fábrica a través de la clase `Html` [API], que dispone de métodos para convertir cualquier objeto `Spanned` en su representación HTML equivalente. Veamos cómo. Recuperemos el texto de la ventana anterior y utilicemos el método `Html.toHtml(Spannable)` para convertirlo a formato HTML:

```
//Obtiene el texto del control con etiquetas de formato HTML
String aux2 = Html.toHtml(txtTexto.getText());
```

Haciendo esto, obtendríamos una cadena de texto como la siguiente, que ya podríamos por ejemplo almacenar en una base de datos o publicar en cualquier web sin perder el formato de texto establecido:

```
<p>Esto es un <b>simulacro</b>.</p>
```

La operación contraria también es posible, es decir, cargar un cuadro de texto de Android (`EditText`) o una etiqueta (`TextView`) a partir de un fragmento de texto en formato HTML. Para ello podemos utilizar el método `Html.fromHtml(String)` de la siguiente forma:

```
//Asigna texto con formato HTML
txtTexto.setText(
Html.fromHtml("<p>Esto es un <b>simulacro</b>.</p>"),
BufferType.SPANNABLE);
```

Desgraciadamente, aunque es de agradecer que este trabajo venga hecho de casa, hay que decir que tan sólo funciona de forma completa con las opciones de formato más básicas,

como negritas, cursivas, subrayado o colores de texto, quedando no soportadas otras sorprendentemente básicas como el tamaño del texto, que aunque sí es correctamente traducido por el método `toHtml()`, es descartado por el método contrario `fromHtml()`. Sí se soporta la inclusión de imágenes, aunque esto lo dejamos para más adelante ya que requiere algo más de explicación.

> El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-cuadrosdetexto.zip

Checkboxes y RadioButtons

Tras hablar de varios de los controles indispensables en cualquier aplicación Android, como son los botones y los cuadros de texto, en este nuevo apartado vamos a ver cómo utilizar otros dos tipos de controles básicos en muchas aplicaciones, los *checkboxes* y los *radio buttons*.

Control CheckBox [API]

Un control *checkbox* se suele utilizar para marcar o desmarcar opciones en una aplicación, y en Android está representado por la clase del mismo nombre, `CheckBox`. La forma de definirlo en nuestra interfaz y los métodos disponibles para manipularlos desde nuestro código son análogos a los ya comentados para el control `ToggleButton`. De esta forma, para definir un control de este tipo en nuestro layout podemos utilizar el código siguiente, que define un checkbox con el texto "*Márcame*":

```
<CheckBox android:id="@+id/ChkMarcame"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Márcame!" />
```

En cuanto a la personalización del control podemos decir que éste extiende [indirectamente] del control `TextView`, por lo que todas las opciones de formato ya comentadas en capítulos anteriores son válidas también para este control.

En el código de la aplicación podremos hacer uso de los métodos `isChecked()` para conocer el estado del control, y `setChecked(estado)` para establecer un estado concreto para el control.

```
if (checkBox.isChecked()) {
    checkBox.setChecked(false);
}
```

En cuanto a los posibles eventos que puede lanzar este control, el más interesante es sin duda el que informa de que ha cambiado el estado del control, que recibe el nombre de `onCheckedChanged`. Para implementar las acciones de este evento podríamos utilizar por tanto la siguiente lógica:

```
final CheckBox cb = (CheckBox)findViewById(R.id.chkMarcame);

cb.setOnCheckedChangeListener(
    new CheckBox.OnCheckedChangeListener() {
        public void onCheckedChanged(CompoundButton buttonView,
                                          boolean isChecked) {
            if (isChecked) {
                cb.setText("Checkbox marcado!");
            }
            else {
                cb.setText("Checkbox desmarcado!");
        }
    }
});
```

Control RadioButton [API]

Al igual que los controles *checkbox*, un *radio button* puede estar marcado o desmarcado, pero en este caso suelen utilizarse dentro de un grupo de opciones donde una, y sólo una, de ellas debe estar marcada obligatoriamente, es decir, que si se marca una de ellas se desmarcará automáticamente la que estuviera activa anteriormente. En Android, un grupo de botones `RadioButton` se define mediante un elemento `RadioGroup`, que a su vez contendrá todos los elementos `RadioButton` necesarios. Veamos un ejemplo de cómo definir un grupo de dos controles `RadioButton` en nuestra interfaz:

```
<RadioGroup android:id="@+id/gruporb"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >

    <RadioButton android:id="@+id/radio1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Opción 1" />

    <RadioButton android:id="@+id/radio2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Opción 2" />

</RadioGroup>
```

En primer lugar vemos cómo podemos definir el grupo de controles indicando su orientación (vertical u horizontal) al igual que ocurría por ejemplo con un `LinearLayout`. Tras esto, se añaden todos los objetos `RadioButton` necesarios indicando su ID mediante la propiedad `android:id` y su texto mediante `android:text`.

Una vez definida la interfaz podremos manipular el control desde nuestro código java haciendo uso de los diferentes métodos del control `RadioGroup`, los más importantes: `check(id)` para marcar una opción determinada mediante su ID, `clearCheck()` para desmarcar todas las opciones, y `getCheckedRadioButtonId()` que como su nombre indica devolverá el ID de la opción marcada (o el valor -1 si no hay ninguna marcada). Veamos un ejemplo:

```
final RadioGroup rg = (RadioGroup)findViewById(R.id.gruporb);
rg.clearCheck();
rg.check(R.id.radio1);
int idSeleccionado = rg.getCheckedRadioButtonId();
```

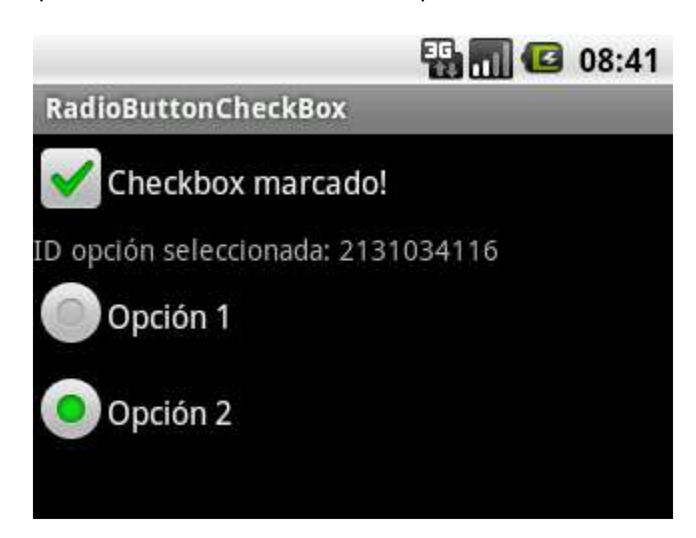
En cuanto a los eventos lanzados, al igual que en el caso de los checkboxes, el más importante será el que informa de los cambios en el elemento seleccionado, llamado también en este caso `onCheckedChange`.

Vemos cómo tratar este evento del objeto `RadioGroup`:

```
final RadioGroup rg = (RadioGroup)findViewById(R.id.gruporb);

rg.setOnCheckedChangeListener(
    new RadioGroup.OnCheckedChangeListener() {
        public void onCheckedChanged(RadioGroup group, int checkedId) {
            lblMensaje.setText("ID opcion seleccionada: " + checkedid);
        }
});
```

Veamos finalmente una imagen del aspecto de estos dos nuevos tipos de controles básicos que hemos comentado en este apartado:



El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-radio-check.zip

Listas Desplegables

Una vez repasados los controles básicos que podemos utilizar en nuestras aplicaciones Android, vamos a dedicar los próximos apartados a describir los diferentes controles de selección disponibles en la plataforma. Al igual que en otros frameworks Android dispone de diversos controles que nos permiten seleccionar una opción dentro de una lista de posibilidades. Así, podremos utilizar listas desplegables (`Spinner`), listas fijas (`ListView`),

tablas (`GridView`) y otros controles específicos de la plataforma como por ejemplo las galerías de imágenes (`Gallery`).

En este primer tema dedicado a los controles de selección vamos a describir un elemento importante y común a todos ellos, los adaptadores, y lo vamos a aplicar al primer control de los indicados, las listas desplegables.

Adaptadores en Android (adapters)

Para los desarrolladores de java que hayan utilizado frameworks de interfaz gráfica como Swing, el concepto de adaptador les resultará familiar. Un adaptador representa algo así como una interfaz común al modelo de datos que existe por detrás de todos los controles de selección que hemos comentado. Dicho de otra forma, todos los controles de selección accederán a los datos que contienen a través de un adaptador.

Además de proveer de datos a los controles visuales, el adaptador también será responsable de generar a partir de estos datos las vistas específicas que se mostrarán dentro del control de selección. Por ejemplo, si cada elemento de una lista estuviera formado a su vez por una imagen y varias etiquetas, el responsable de generar y establecer el contenido de todos estos "sub-elementos" a partir de los datos será el propio adaptador.

Android proporciona de serie varios tipos de adaptadores sencillos, aunque podemos extender su funcionalidad fácilmente para adaptarlos a nuestras necesidades. Los más comunes son los siguientes:

- `ArrayAdapter`. Es el más sencillo de todos los adaptadores, y provee de datos a un control de selección a partir de un array de objetos de cualquier tipo.
- `SimpleAdapter`. Se utiliza para mapear datos sobre los diferentes controles definidos en un fichero XML de layout.
- `SimpleCursorAdapter`. Se utiliza para mapear las columnas de un cursor sobre los diferentes elementos visuales contenidos en el control de selección.

Para no complicar excesivamente los tutoriales, por ahora nos vamos a conformar con describir la forma de utilizar un `ArrayAdapter` con los diferentes controles de selección disponibles. Más adelante aprenderemos a utilizar el resto de adaptadores en contextos más específicos.

Veamos cómo crear un adaptador de tipo `ArrayAdapter` para trabajar con un array genérico de java:

```
final String[] datos =
        new String[]{"Elem1","Elem2","Elem3","Elem4","Elem5"};

ArrayAdapter<String> adaptador =
        new ArrayAdapter<String>(this,
            android.R.layout.simple_spinner_item, datos);
```

Comentemos un poco el código. Sobre la primera línea no hay nada que decir, es tan sólo la definición del array java que contendrá los datos a mostrar en el control, en este caso un array

sencillo con cinco cadenas de caracteres. En la segunda línea creamos el adaptador en sí, al que pasamos 3 parámetros:

1. El contexto, que normalmente será simplemente una referencia a la actividad donde se crea el adaptador.
2. El ID del layout sobre el que se mostrarán los datos del control. En este caso le pasamos el ID de un layout predefinido en Android (`android.R.layout.simple_spinner_item`), formado únicamente por un control `TextView`, pero podríamos pasarle el ID de cualquier layout de nuestro proyecto con cualquier estructura y conjunto de controles, más adelante veremos cómo.
3. El array que contiene los datos a mostrar.

Con esto ya tendríamos creado nuestro adaptador para los datos a mostrar y ya tan sólo nos quedaría asignar este adaptador a nuestro control de selección para que éste mostrase los datos en la aplicación.

Control Spinner [API]

Las listas desplegables en Android se llaman `Spinner`. Funcionan de forma similar al de cualquier control de este tipo, el usuario selecciona la lista, se muestra una especie de lista emergente al usuario con todas las opciones disponibles y al seleccionarse una de ellas ésta queda fijada en el control. Para añadir una lista de este tipo a nuestra aplicación podemos utilizar el código siguiente:

```
<Spinner android:id="@+id/CmbOpciones"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
```

Poco vamos a comentar de aquí ya que lo que nos interesan realmente son los datos a mostrar. En cualquier caso, las opciones para personalizar el aspecto visual del control (fondo, color y tamaño de fuente, etc) son las mismas ya comentadas para los controles básicos.

Para enlazar nuestro adaptador (y por tanto nuestros datos) a este control utilizaremos el siguiente código java:

```
final Spinner cmbOpciones = (Spinner)findViewById(R.id.CmbOpciones);

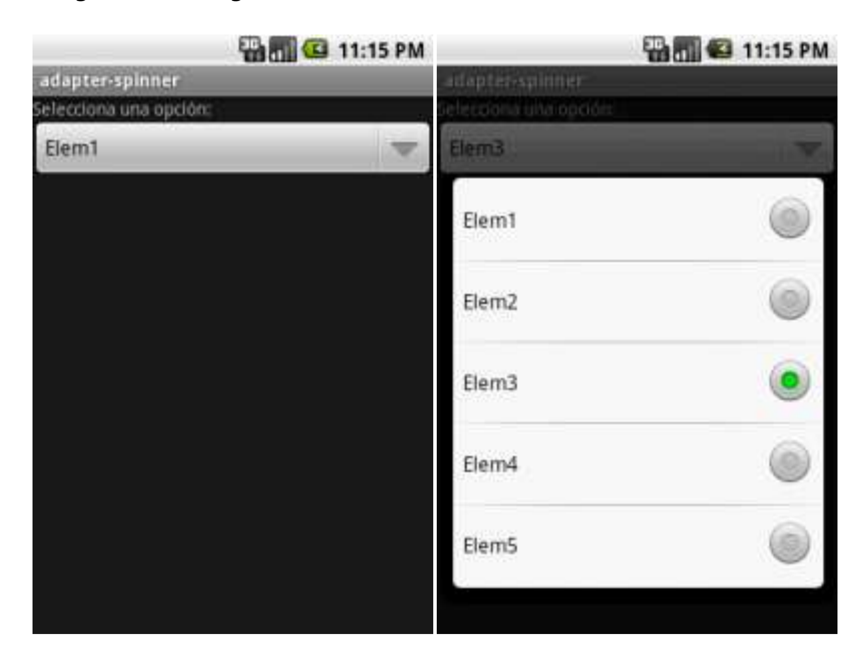
adaptador.setDropDownViewResource(
        android.R.layout.simple_spinner_dropdown_item);

cmbOpciones.setAdapter(adaptador);
```

Comenzamos como siempre por obtener una referencia al control a través de su ID. Y en la última línea asignamos el adaptador al control mediante el método `setAdapter()`. ¿Y la segunda línea para qué es? Cuando indicamos en el apartado anterior cómo construir un adaptador vimos cómo uno de los parámetros que le pasábamos era el ID del layout que utilizaríamos para visualizar los elementos del control. Sin embargo, en el caso del control `Spinner`, este layout tan sólo se aplicará al elemento seleccionado en la lista, es decir, al que

se muestra directamente sobre el propio control cuando no está desplegado. Sin embargo, antes indicamos que el funcionamiento normal del control Spinner incluye entre otras cosas mostrar una lista emergente con todas las opciones disponibles. Pues bien, para personalizar también el aspecto de cada elemento en dicha lista emergente tenemos el método setDropDownViewResource(ID_layout), al que podemos pasar otro ID de layout distinto al primero sobre el que se mostrarán los elementos de la lista emergente. En este caso hemos utilizado otro layout predefinido en Android para las listas desplegables (android.R.layout.simple_spinner_dropdown_item).

Con estas simples líneas de código conseguiremos mostrar un control como el que vemos en las siguientes imágenes:



Como se puede observar en las imágenes, la representación del elemento seleccionado (primera imagen) y el de las opciones disponibles (segunda imagen) es distinto, incluyendo el segundo de ellos incluso algún elemento gráfico a la derecha para mostrar el estado de cada opción. Como hemos comentado, esto es debido a la utilización de dos layouts diferentes para uno y otros elementos.

En cuanto a los eventos lanzados por el control Spinner, el más comúnmente utilizado será el generado al seleccionarse una opción de la lista desplegable, onItemSelected.

Para capturar este evento se procederá de forma similar a lo ya visto para otros controles anteriormente, asignándole su controlador mediante el método setOnItemSelectedListener():

```
cmbOpciones.setOnItemSelectedListener(
        new AdapterView.OnItemSelectedListener() {
        public void onItemSelected(AdapterView<?> parent,
            android.view.View v, int position, long id) {
                lblMensaje.setText("Seleccionado: " + datos[position]);
        }


        public void onNothingSelected(AdapterView<?> parent) {
```

```
            lblMensaje.setText("");
        }
});
```

Para este evento definimos dos métodos, el primero de ellos (onItemSelected) que será llamado cada vez que se selecciones una opción en la lista desplegable, y el segundo (onNothingSelected) que se llamará cuando no haya ninguna opción seleccionada (esto puede ocurrir por ejemplo si el adaptador no tiene datos).

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-spinner.zip

Listas

En el apartado anterior ya comenzamos a hablar de los controles de selección en Android, empezando por explicar el concepto de adaptador y describiendo el control Spinner. En este nuevo apartado nos vamos a centrar en el control de selección más utilizado de todos, el ListView.

Un control ListView muestra al usuario una lista de opciones seleccionables directamente sobre el propio control, sin listas emergentes como en el caso del control Spinner. En caso de existir más opciones de las que se pueden mostrar sobre el control se podrá por supuesto hacer *scroll* sobre la lista para acceder al resto de elementos.

Veamos primero cómo podemos añadir un control ListView a nuestra interfaz de usuario:

```
<ListView android:id="@+id/LstOpciones"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

Una vez más, podremos modificar el aspecto del control utilizando las propiedades de fuente y color ya comentadas en capítulos anteriores. Por su parte, para enlazar los datos con el control podemos utilizar por ejemplo el mismo código que ya vimos para el control Spinner. Definiremos primero un array con nuestros datos de prueba, crearemos posteriormente el adaptador de tipo ArrayAdapter y lo asignaremos finalmente al control mediante el método setAdapter():

```
final String[] datos =
    new String[]{"Elem1","Elem2","Elem3","Elem4","Elem5"};

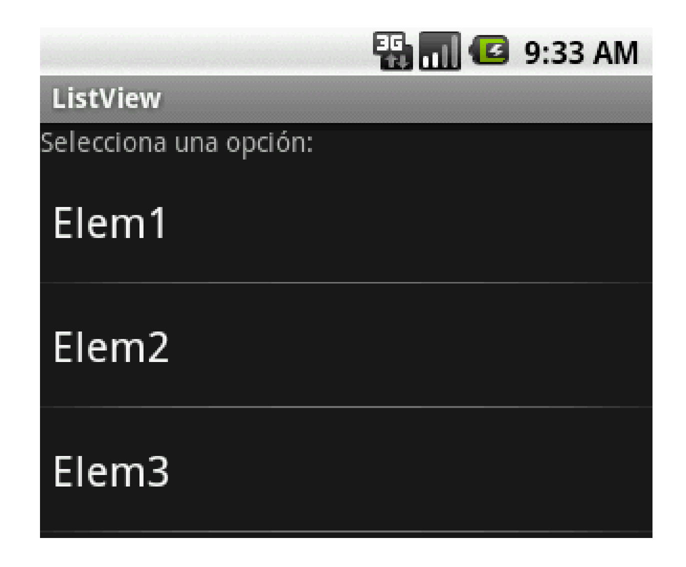
ArrayAdapter<String> adaptador =
    new ArrayAdapter<String>(this,
        android.R.layout.simple_list_item_1, datos);

ListView lstOpciones = (ListView)findViewById(R.id.LstOpciones);

lstOpciones.setAdapter(adaptador);
```

En este caso, para mostrar los datos de cada elemento hemos utilizado otro layout genérico de Android para los controles de tipo ListView

(`android.R.layout.simple_list_item_1`), formado únicamente por un `TextView` con unas dimensiones determinadas. La lista creada quedaría como se muestra en la imagen siguiente:



Como podéis comprobar el uso básico del control `ListView` es completamente análogo al ya comentado para el control `Spinner`.

Si quisiéramos realizar cualquier acción al pulsarse sobre un elemento de la lista creada tendremos que implementar el evento `onItemClick`. Veamos cómo con un ejemplo:

```
lstOpciones.setOnItemClickListener(new OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> a, View v, int position, long id) {
        //Acciones necesarias al hacer click
    }
});
```

Hasta aquí todo sencillo. Pero, ¿y si necesitamos mostrar datos más complejos en la lista? ¿qué ocurre si necesitamos que cada elemento de la lista esté formado a su vez por varios elementos? Pues vamos a provechar este capítulo dedicado a los `ListView` para ver cómo podríamos conseguirlo, aunque todo lo que comentaré es extensible a otros controles de selección.

Para no complicar mucho el tema vamos a hacer que cada elemento de la lista muestre por ejemplo dos líneas de texto a modo de título y subtítulo con formatos diferentes (por supuesto se podrían añadir muchos más elementos, por ejemplo imágenes, checkboxes, etc).

En primer lugar vamos a crear una nueva clase java para contener nuestros datos de prueba. Vamos a llamarla `Titular` y tan sólo va a contener dos atributos, título y subtítulo.

```
package net.sgoliver;

public class Titular
```

```
{
    private String titulo;
    private String subtitulo;

    public Titular(String tit, String sub){
        titulo = tit;
        subtitulo = sub;
    }

    public String getTitulo(){
        return titulo;
    }

    public String getSubtitulo(){
        return subtitulo;
    }
}
```

En cada elemento de la lista queremos mostrar ambos datos, por lo que el siguiente paso será crear un layout XML con la estructura que deseemos. En mi caso voy a mostrarlos en dos etiquetas de texto (`TextView`), la primera de ellas en negrita y con un tamaño de letra un poco mayor.

Llamaremos a este layout "`listitem_titular.xml`":

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:orientation="vertical">

<TextView android:id="@+id/LblTitulo"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textStyle="bold"
    android:textSize="20px" />

<TextView android:id="@+id/LblSubTitulo"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textStyle="normal"
    android:textSize="12px" />

</LinearLayout>
```

Ahora que ya tenemos creados tanto el soporte para nuestros datos como el layout que necesitamos para visualizarlos, lo siguiente que debemos hacer será indicarle al adaptador cómo debe utilizar ambas cosas para generar nuestra interfaz de usuario final. Para ello vamos a crear nuestro propio adaptador extendiendo de la clase `ArrayAdapter`.

```
class AdaptadorTitulares extends ArrayAdapter {

    Activity context;
```

```
        AdaptadorTitulares(Activity context) {
            super(context, R.layout.listitem_titular, datos);
            this.context = context;
        }

        public View getView(int position, View convertView, ViewGroup parent)
{
        LayoutInflater inflater = context.getLayoutInflater();
        View item = inflater.inflate(R.layout.listitem_titular, null);

        TextView lblTitulo = (TextView)item.findViewById(R.id.LblTitulo);
        lblTitulo.setText(datos[position].getTitulo());

        TextView lblSubtitulo =
(TextView)item.findViewById(R.id.LblSubTitulo);
        lblSubtitulo.setText(datos[position].getSubtitulo());

        return(item);
    }
}
```

Analicemos el código anterior. Lo primero que encontramos es el constructor para nuestro adaptador, al que sólo pasaremos el contexto (que será la actividad desde la que se crea el adaptador). En este constructor tan sólo guardaremos el contexto para nuestro uso posterior y llamaremos al constructor padre tal como ya vimos al principio de este capítulo, pasándole el ID del layout que queremos utilizar (en nuestro caso el nuevo que hemos creado, `listitem_titular`) y el array que contiene los datos a mostrar.

Posteriormente, redefinimos el método encargado de generar y rellenar con nuestros datos todos los controles necesarios de la interfaz gráfica de cada elemento de la lista. Este método es `getView()`.

El método `getView()` se llamará cada vez que haya que mostrar un elemento de la lista. Lo primero que debe hacer es "inflar" el layout XML que hemos creado. Esto consiste en consultar el XML de nuestro layout y crear e inicializar la estructura de objetos java equivalente. Para ello, crearemos un nuevo objeto `LayoutInflater` y generaremos la estructura de objetos mediante su método `inflate(id_layout)`.

Tras esto, tan sólo tendremos que obtener la referencia a cada una de nuestras etiquetas como siempre lo hemos hecho y asignar su texto correspondiente según los datos de nuestro array y la posición del elemento actual (parámetro position del método `getView()`).

Una vez tenemos definido el comportamiento de nuestro adaptador la forma de proceder en la actividad principal será análoga a lo ya comentado, definiremos el array de datos de prueba, crearemos el adaptador y lo asignaremos al control mediante `setAdapter()`:

```
private Titular[] datos =
    new Titular[]{
        new Titular("Título 1", "Subtítulo largo 1"),
        new Titular("Título 2", "Subtítulo largo 2"),
```

```
        new Titular("Título 3", "Subtítulo largo 3"),
        new Titular("Título 4", "Subtítulo largo 4"),
        new Titular("Título 5", "Subtítulo largo 5")};

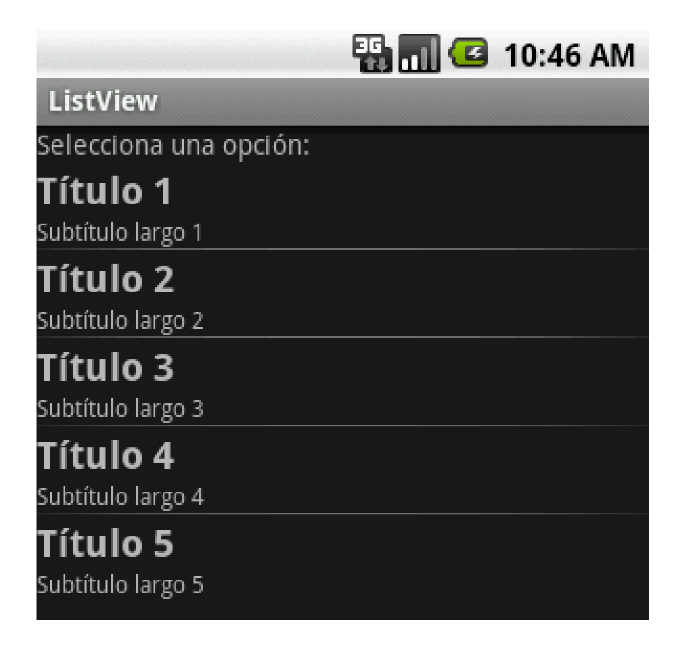
//...
//...

AdaptadorTitulares adaptador =
        new AdaptadorTitulares(this);

ListView lstOpciones = (ListView)findViewById(R.id.LstOpciones);

lstOpciones.setAdapter(adaptador);
```

Hecho esto, y si todo ha ido bien, nuestra nueva lista debería quedar como vemos en la imagen siguiente:



Aunque ya sabemos utilizar y personalizar las listas en Android, en el próximo apartado daremos algunas indicaciones para utilizar de una forma mucho más eficiente los controles de este tipo, algo que los usuarios de nuestra aplicación agradecerán enormemente al mejorarse la respuesta de la aplicación y reducirse el consumo de batería.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-listview.zip

Optimización de listas

En el apartado anterior ya vimos cómo utilizar los controles de tipo `ListView` en Android. Sin embargo, acabamos comentando que existía una forma más eficiente de hacer uso de dicho control, de forma que la respuesta de nuestra aplicación fuera más ágil y se redujese el consumo de batería, algo que en plataformas móviles siempre es importante.

Como base para este tema vamos a utilizar como código el que ya escribimos en el capítulo anterior, por lo que si has llegado hasta aquí directamente te recomiendo que leas primero el primer post dedicado al control `ListView`.

Cuando comentamos cómo crear nuestro propio adaptador, extendiendo de `ArrayAdapter`, para personalizar la forma en que nuestros datos se iban a mostrar en la lista escribimos el siguiente código:

```
class AdaptadorTitulares extends ArrayAdapter {
    Activity context;

    AdaptadorTitulares(Activity context) {
        super(context, R.layout.listitem_titular, datos);
        this.context = context;
    }

    public View getView(int position, View convertView, ViewGroup parent) {
        LayoutInflater inflater = context.getLayoutInflater();
        View item = inflater.inflate(R.layout.listitem_titular, null);

        TextView lblTitulo = (TextView)item.findViewById(R.id.LblTitulo);
        lblTitulo.setText(datos[position].getTitulo());

        TextView lblSubtitulo =
                     (TextView)item.findViewById(R.id.LblSubTitulo);
        lblSubtitulo.setText(datos[position].getSubtitulo());

        return(item);
    }
}
```

Centrándonos en la definición del método `getView()` vimos que la forma normal de proceder consistía en primer lugar en "inflar" nuestro layout XML personalizado para crear todos los objetos correspondientes (con la estructura descrita en el XML) y posteriormente acceder a dichos objetos para modificar sus propiedades. Sin embargo, hay que tener en cuenta que esto se hace todas y cada una de las veces que se necesita mostrar un elemento de la lista en pantalla, se haya mostrado ya o no con anterioridad, ya que Android no "guarda" los elementos de la lista que desaparecen de pantalla (por ejemplo al hacer *scroll* sobre la lista). El efecto de esto es obvio, dependiendo del tamaño de la lista y sobre todo de la complejidad del layout que hayamos definido esto puede suponer la creación y destrucción de cantidades ingentes de objetos (que puede que ni siquiera nos sean necesarios), es decir, que la acción de inflar un layout XML puede ser bastante costosa, lo que podría aumentar mucho, y sin necesidad, el uso de CPU, de memoria, y de batería.

Para aliviar este problema, Android nos propone un método que permite reutilizar algún layout que ya hayamos inflado con anterioridad y que ya no nos haga falta por algún motivo, por ejemplo porque el elemento correspondiente de la lista ha desaparecido de la pantalla al hacer *scroll*. De esta forma evitamos todo el trabajo de crear y estructurar todos los objetos asociados al layout, por lo que tan sólo nos quedaría obtener la referencia a ellos mediante `findViewById()` y modificar sus propiedades.

¿Pero cómo podemos reutilizar estos layouts "obsoletos"? Pues es bien sencillo, siempre que exista algún layout que pueda ser reutilizado éste se va a recibir a través del parámetro

convertView del método getView(). De esta forma, en los casos en que éste no sea null podremos obviar el trabajo de inflar el layout. Veamos cómo quedaría el método getView() tras esta optimización:

```
public View getView(int position, View convertView, ViewGroup parent)
{
    View item = convertView;

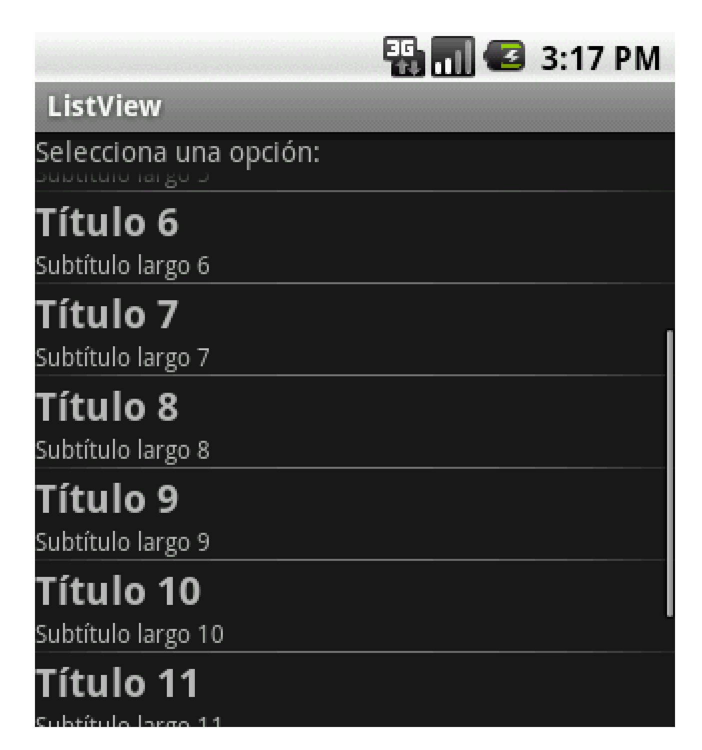
    if(item == null)
    {
        LayoutInflater inflater = context.getLayoutInflater();
        item = inflater.inflate(R.layout.listitem_titular, null);
    }

    TextView lblTitulo = (TextView)item.findViewById(R.id.LblTitulo);
    lblTitulo.setText(datos[position].getTitulo());

    TextView lblSubtitulo = (TextView)item.findViewById(R.id.LblSubTitulo);
    lblSubtitulo.setText(datos[position].getSubtitulo());

    return(item);
}
```

Si añadimos más elementos a la lista y ejecutamos ahora la aplicación podemos comprobar que al hacer scroll sobre la lista todo sigue funcionando con normalidad, con la diferencia de que le estamos ahorrando gran cantidad de trabajo a la CPU.



Pero vamos a ir un poco más allá. Con la optimización que acabamos de implementar conseguimos ahorrarnos el trabajo de inflar el layout definido cada vez que se muestra un nuevo elemento. Pero aún hay otras dos llamadas relativamente costosas que se siguen ejecutando en todas las llamadas. Me refiero a la obtención de la referencia a cada uno de los objetos a modificar mediante el método findViewById(). La búsqueda por ID de un control determinado dentro del árbol de objetos de un layout también puede ser una tarea costosa dependiendo de la complejidad del propio layout. ¿Por qué no aprovechamos que

estamos “guardando” un layout anterior para guardar también la referencia a los controles que lo forman de forma que no tengamos que volver a buscarlos? Pues eso es exactamente lo que vamos a hacer mediante lo que en los ejemplos de Android llaman un `ViewHolder`. La clase `ViewHolder` tan sólo va a contener una referencia a cada uno de los controles que tengamos que manipular de nuestro layout, en nuestro caso las dos etiquetas de texto. Definamos por tanto esta clase de la siguiente forma:

```
static class ViewHolder {
    TextView titulo;
    TextView subtitulo; }
```

La idea será por tanto crear e inicializar el objeto `ViewHolder` la primera vez que inflemos un elemento de la lista y asociarlo a dicho elemento de forma que posteriormente podamos recuperarlo fácilmente. ¿Pero dónde lo guardamos? Fácil, en Android todos los controles tienen una propiedad llamada `Tag` (podemos asignarla y recuperarla mediante los métodos `setTag()` y `getTag()` respectivamente) que puede contener cualquier tipo de objeto, por lo que resulta ideal para guardar nuestro objeto `ViewHolder`. De esta forma, cuando el parámetro `convertView` llegue informado sabremos que también tendremos disponibles las referencias a sus controles hijos a través de la propiedad `Tag`. Veamos el código modificado de `getView()` para aprovechar esta nueva optimización:

```
public View getView(int position, View convertView, ViewGroup parent)
{
    View item = convertView;
    ViewHolder holder;

    if(item == null)
    {
        LayoutInflater inflater = context.getLayoutInflater();
        item = inflater.inflate(R.layout.listitem_titular, null);

        holder = new ViewHolder();
        holder.titulo = (TextView)item.findViewById(R.id.LblTitulo);
        holder.subtitulo = (TextView)item.findViewById(R.id.LblSubTitulo);

        item.setTag(holder);
    }
    else
    {
        holder = (ViewHolder)item.getTag();
    }

    holder.titulo.setText(datos[position].getTitulo());
    holder.subtitulo.setText(datos[position].getSubtitulo());

    return(item);
}
```

Con estas dos optimizaciones hemos conseguido que la aplicación sea mucho más respetuosa con los recursos del dispositivo de nuestros usuarios, algo que sin duda nos agradecerán.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-listview-optimizado.zip

Grids

Tras haber visto en apartados anteriores los dos controles de selección más comunes en cualquier interfaz gráfica, como son las listas desplegables (Spinner) y las listas "fijas" (ListView), tanto en su versión básica como optimizada, en este nuevo apartado vamos a terminar de comentar los controles de selección con otro menos común pero no por ello menos útil, el control GridView.

El control GridView de Android presenta al usuario un conjunto de opciones seleccionables distribuidas de forma tabular, o dicho de otra forma, divididas en filas y columnas. Dada la naturaleza del control ya podéis imaginar sus propiedades más importantes, que paso a enumerar a continuación:

Propiedad	Descripción
android:numColumns	Indica el número de columnas de la tabla o "auto_fit" si queremos que sea calculado por el propio sistema operativo a partir de las siguientes propiedades.
android:columnWidth	Indica el ancho de las columnas de la tabla.
android:horizontalSpacing	Indica el espacio horizontal entre celdas.
android:verticalSpacing	Indica el espacio vertical entre celdas.
android:stretchMode	Indica qué hacer con el espacio horizontal sobrante. Si se establece al valor columnWidth este espacio será absorbido a partes iguales por las columnas de la tabla. Si por el contrario se establece a spacingWidth será absorbido a partes iguales por los espacios entre celdas.

Veamos cómo definiríamos un GridView de ejemplo en nuestra aplicación:

```
<GridView android:id="@+id/GridOpciones"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:numColumns="auto_fit"
    android:columnWidth="80px"
    android:horizontalSpacing="5px"
    android:verticalSpacing="10px"
    android:stretchMode="columnWidth" />
```

Una vez definida la interfaz de usuario, la forma de asignar los datos desde el código de la aplicación es completamente análoga a la ya comentada tanto para las listas desplegables como para las listas estáticas: creamos un array genérico que contenga nuestros datos de prueba, declaramos un adaptador de tipo ArrayAdapter pasándole en este caso un layout genérico (simple_list_item_1, compuesto por un simple TextView) y asociamos el adaptador al control GridView mediante su método setAdapter():

```
private String[] datos = new String[25];
//...
for(int i=1; i<=25; i++)
        datos[i-1] = "Dato " + i;

ArrayAdapter<String> adaptador =
        new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1,
datos);

final GridView grdOpciones = (GridView)findViewById(R.id.GridOpciones);

grdOpciones.setAdapter(adaptador);
```

Por defecto, los datos del *array* se añadirán al control `GridView` ordenados por filas, y por supuesto, si no caben todos en la pantalla se podrá hacer *scroll* sobre la tabla. Vemos en una imagen cómo queda nuestra aplicación de prueba:

En cuanto a los eventos disponibles, el más interesante vuelve a ser el lanzado al seleccionarse una celda determinada de la tabla: `onItemSelected`. Este evento podemos capturarlo de la misma forma que hacíamos con los controles `Spinner` y `ListView`. Veamos un ejemplo de cómo hacerlo:

```
grdOpciones.setOnItemSelectedListener(
    new AdapterView.OnItemSelectedListener() {
            public void onItemSelected(AdapterView<?> parent,
                android.view.View v, int position, long id) {
                    lblMensaje.setText("Seleccionado: " + datos[position]);
            }

            public void onNothingSelected(AdapterView<?> parent) {
                lblMensaje.setText("");
            }
});
```

Todo lo comentado hasta el momento se refiere al uso básico del control `GridView`, pero por supuesto podríamos aplicar de forma prácticamente directa todo lo comentado para las listas en los dos apartados anteriores, es decir, la personalización de las celdas para presentar datos complejos creando nuestro propio adaptador, y las distintas optimizaciones para mejorar el rendimiento de la aplicación y el gasto de batería.

Y con esto finalizamos todo lo que tenía previsto contar sobre los distintos controles disponibles "de serie" en Android para construir nuestras interfaces de usuario. Existen muchos más, y es posible que los comentemos más adelante, pero por el momento nos vamos a conformar con los ya vistos. En el próximo apartado, y para terminar con la serie dedicada a los controles Android, veremos las distintas formas de crear controles de usuario personalizados.

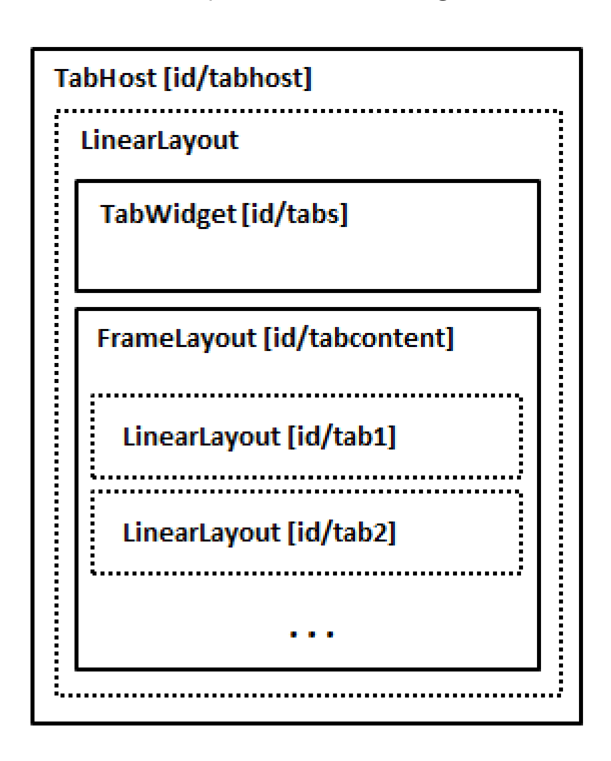
> El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-gridview.zip

Pestañas

En apartados anteriores hemos visto como dar forma a la interfaz de usuario de nuestra aplicación mediante el uso de diversos tipos de *layouts*, como por ejemplo los lineales, absolutos, relativos, u otros más elaborados como los de tipo lista o tabla. Éstos van a ser siempre los elementos organizativos básicos de nuestra interfaz, pero sin embargo, dado el poco espacio con el que contamos en las pantallas de muchos dispositivos, o simplemente por cuestiones de organización, a veces es necesario/interesante dividir nuestros controles en varias pantallas. Y una de las formas clásicas de conseguir esto consiste en la distribución de los elementos por pestañas o *tabs*. Android por supuesto permite utilizar este tipo de interfaces, aunque lo hace de una forma un tanto peculiar, ya que la implementación no va a depender de un sólo elemento sino de varios, que además deben estar distribuidos y estructurados de una forma determinada nada arbitraria. Adicionalmente no nos bastará simplemente con definir la interfaz en XML como hemos hecho en otras ocasiones, sino que también necesitaremos completar el conjunto con algunas líneas de código. Desarrollemos esto poco a poco.

En Android, el elemento principal de un conjunto de pestañas será el control `TabHost`. Éste va a ser el contenedor principal de nuestro conjunto de pestañas y deberá tener obligatoriamente como id el valor "`@android:id/tabhost`". Dentro de éste vamos a incluir un `LinearLayout` que nos servirá para distribuir verticalmente las secciones

principales del layout: la sección de pestañas en la parte superior y la sección de contenido en la parte inferior. La sección de pestañas se representará mediante un elemento `TabWidget`, que deberá tener como id el valor "`@android:id/tabs`", y como contenedor para el contenido de las pestañas añadiremos un `FrameLayout` con el id obligatorio "`@android:id/tabcontent`". Por último, dentro del `FrameLayout` incluiremos el contenido de cada pestaña, normalmente cada uno dentro de su propio layout principal (en mi caso he utilizado `LinearLayout`) y con un id único que nos permita posteriormente hacer referencia a ellos fácilmente (en mi caso he utilizado por ejemplo los ids "*tab1*", "*tab2*", …). A continuación represento de forma gráfica toda la estructura descrita.



Si traducimos esta estructura a nuestro fichero de layout XML tendríamos lo siguiente:

```
<TabHost android:id="@android:id/tabhost"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >

         <TabWidget android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:id="@android:id/tabs" />

         <FrameLayout android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:id="@android:id/tabcontent" >

            <LinearLayout android:id="@+id/tab1"
                android:orientation="vertical"
                android:layout_width="match_parent"
                android:layout_height="match_parent" >
                <TextView android:id="@+id/textView1"
                    android:text="Contenido Tab 1"
```

```
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content" />
        </LinearLayout>

            <LinearLayout android:id="@+id/tab2"
                android:orientation="vertical"
                android:layout_width="match_parent"
                android:layout_height="match_parent" >
                <TextView android:id="@+id/textView2"
                    android:text="Contenido Tab 2"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content" />
        </LinearLayout>
         </FrameLayout>
    </LinearLayout>

</TabHost>
```

Como podéis ver, como contenido de las pestañas tan sólo he añadido por simplicidad una etiqueta de texto con el texto "*Contenido Tab NºTab*". Esto nos permitirá ver que el conjunto de pestañas funciona correctamente cuando ejecutemos la aplicación.

Con esto ya tendríamos montada toda la estructura de controles necesaria para nuestra interfaz de pestañas. Sin embargo, como ya dijimos al principio del apartado, con esto no es suficiente. Necesitamos asociar de alguna forma cada pestaña con su contenido, de forma que el control se comporte correctamente cuando cambiamos de pestaña. Y esto tendremos que hacerlo mediante código en nuestra actividad principal.

Empezaremos obteniendo una referencia al control principal `TabHost` y preparándolo para su configuración llamando a su método `setup()`. Tras esto, crearemos un objeto de tipo `TabSpec` para cada una de las pestañas que queramos añadir mediante el método `newTabSpec()`, al que pasaremos como parámetro una etiqueta identificativa de la pestaña (en mi caso de ejemplo "*mitab1*", "*mitab2*", …). Además, también le asignaremos el layout de contenido correspondiente a la pestaña llamando al método `setContent()`, e indicaremos el texto y el icono que queremos mostrar en la pestaña mediante el método `setIndicator(texto, icono)`. Veamos el código completo.

```
Resources res = getResources();

TabHost tabs=(TabHost)findViewById(android.R.id.tabhost);
tabs.setup();

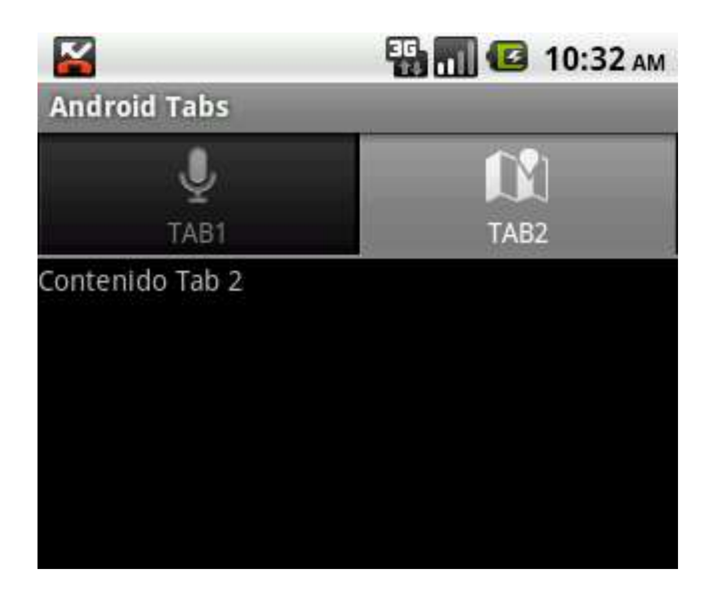
TabHost.TabSpec spec=tabs.newTabSpec("mitab1");
spec.setContent(R.id.tab1);
spec.setIndicator("TAB1",
        res.getDrawable(android.R.drawable.ic_btn_speak_now));
tabs.addTab(spec);

spec=tabs.newTabSpec("mitab2");
spec.setContent(R.id.tab2);
spec.setIndicator("TAB2",
        res.getDrawable(android.R.drawable.ic_dialog_map));
tabs.addTab(spec);

tabs.setCurrentTab(0);
```

Si vemos el código, vemos por ejemplo como para la primera pestaña creamos un objeto `TabSpec` con la etiqueta "*mitab1*", le asignamos como contenido uno de los `LinearLayout` que incluimos en la sección de contenido (en este caso `R.id.tab1`) y finalmente le asignamos el texto "*TAB1*" y el icono `android.R.drawable.ic_btn_speak_now` (Éste es un icono incluido con la propia plataforma Android. Si no existiera en vuestra versión podéis sustituirlo por cualquier otro icono). Finalmente añadimos la nueva pestaña al control mediante el método `addTab()`.

Si ejecutamos ahora la aplicación tendremos algo como lo que muestra la siguiente imagen, donde podremos cambiar de pestaña y comprobar cómo se muestra correctamente el contenido de la misma.



En cuanto a los eventos disponibles del control `TabHost`, aunque no suele ser necesario capturarlos, podemos ver a modo de ejemplo el más interesante de ellos, `OnTabChanged`, que se lanza cada vez que se cambia de pestaña y que nos informa de la nueva pestaña visualizada. Este evento podemos implementarlo y asignarlo mediante el método `setOnTabChangedListener()` de la siguiente forma:

```
tabs.setOnTabChangedListener(new OnTabChangeListener() {
    @Override
    public void onTabChanged(String tabId) {
        Log.i("AndroidTabsDemo", "Pulsada pestaña: " + tabId);
    }
});
```

En el método `onTabChanged()` recibimos como parámetro la etiqueta identificativa de la pestaña (no su ID), que debimos asignar cuando creamos su objeto `TabSpec` correspondiente. Para este ejemplo, lo único que haremos al detectar un cambio de pestaña será escribir en el log de la aplicación un mensaje informativo con la etiqueta de la nueva pestaña visualizada. Así por ejemplo, al cambiar a la segunda pestaña recibiremos el mensaje de log: "*Pulsada pestaña: mitab2*".

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-tabs.zip

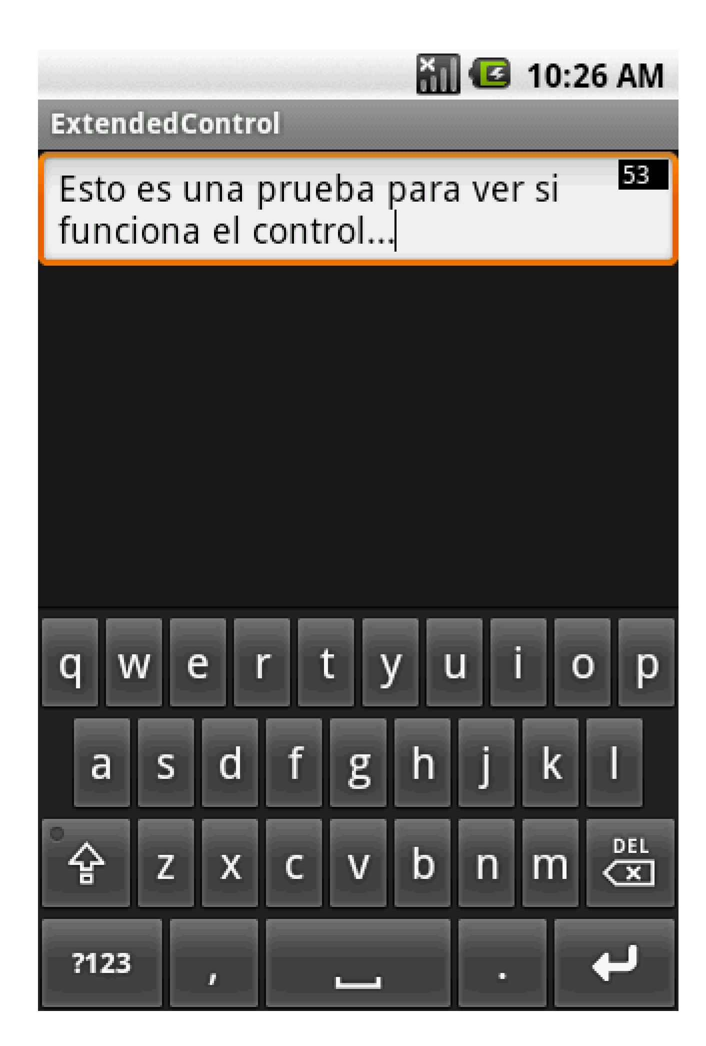
Controles personalizados: Extender controles

En apartados anteriores hemos conocido y aprendido a utilizar muchos de los controles que proporciona Android en su SDK. Con la ayuda de estos controles podemos diseñar interfaces gráficas de lo más variopinto pero en ocasiones, si queremos dar un toque especial y original a nuestra aplicación, o simplemente si necesitamos cierta funcionalidad no presente en los componentes estándar de Android, nos vemos en la necesidad de crear nuestros propios controles personalizados, diseñados a medida de nuestros requisitos.

Android admite por supuesto crear controles personalizados, y permite hacerlo de diferentes formas:

1. Extendiendo la funcionalidad de un control ya existente.
2. Combinando varios controles para formar un otro más complejo.
3. Diseñando desde cero un nuevo control.

En este primer apartado sobre el tema vamos a hablar de la primera opción, es decir, vamos a ver cómo podemos crear un nuevo control partiendo de la base de un control ya existente. A modo de ejemplo, vamos a extender el control `EditText` (cuadro de texto) para que muestre en todo momento el número de caracteres que contiene a medida que se escribe en él. Intentaríamos emular algo así como el editor de mensajes SMS del propio sistema operativo, que nos avisa del número de caracteres que contiene el mensaje. En nuestro caso, como resultado obtendremos un control como el que se muestra en la siguiente imagen:



Como vemos, en la esquina superior derecha del cuadro de texto vamos a mostrar el número de caracteres del mensaje de texto introducido, que irá actualizándose a medida que modificamos el texto. Para empezar, vamos a crear una nueva clase java que extienda del control que queremos utilizar como base, en este caso `EditText`.

```
public class ExtendedEditText extends EditText
{
    //...
}
```

Tras esto, sobrescribiremos siempre los tres constructores heredados, donde por el momento nos limitaremos a llamar al mismo constructor de la clase padre.

```
public ExtendedEditText(Context context, AttributeSet attrs, int defStyle){
    super(context, attrs,defStyle);
}

public ExtendedEditText(Context context, AttributeSet attrs) {
    super(context, attrs);
}

public ExtendedEditText(Context context) {
    super(context);
}
```

Por último el paso más importante. Dado que queremos modificar el aspecto del control para añadir el contador de caracteres tendremos que sobrescribir el evento onDraw(), que es llamado por Android cada vez que hay que redibujar el control en pantalla. Este método recibe como parámetro un objeto Canvas, que no es más que el "lienzo" sobre el que podemos dibujar todos los elementos extra necesarios en el control. El objeto Canvas, proporciona una serie de métodos para dibujar cualquier tipo de elemento (líneas, rectángulos, elipses, texto, bitmaps, …) sobre el espacio ocupado por el control. En nuestro caso tan sólo vamos a necesitar dibujar sobre el control un rectángulo que sirva de fondo para el contador y el texto del contador con el número de caracteres actual del cuadro de texto. No vamos a entrar en muchos detalles sobre la forma de dibujar gráficos ya que ése será tema de otro apartado, pero vamos a ver al menos las acciones principales.

En primer lugar definiremos los "pinceles" (objetos Paint) que utilizaremos para dibujar, uno de ellos (p1) de color negro y relleno sólido para el fondo del contador, y otro (p2) de color blanco para el texto.

```
Paint p1 = new Paint(Paint.ANTI_ALIAS_FLAG);
p1.setColor(Color.BLACK);
p1.setStyle(Style.FILL);

Paint p2 = new Paint(Paint.ANTI_ALIAS_FLAG);
p2.setColor(Color.WHITE);
```

Dado que estos elementos tan sólo hará falta crearlos la primera vez que se dibuje el control, para evitar trabajo innecesario no incluiremos su definición en el método onDraw(), sino que los definiremos como atributos de la clase y los inicializaremos en el constructor del control (en los tres).

Una vez definidos los diferentes pinceles necesarios, dibujaremos el fondo y el texto del contador mediante los métodos drawRect() y drawText(), respectivamente, del objeto canvas recibido en el evento.

```
@Override
public void onDraw(Canvas canvas)
{
    //Llamamos al método de la clase base (EditText)
    super.onDraw(canvas);

    //Dibujamos el fondo negro del contador
    canvas.drawRect(this.getWidth()-30, 5,
        this.getWidth()-5, 20, p1);

    //Dibujamos el número de caracteres sobre el contador
    canvas.drawText("" + this.getText().toString().length(),
        this.getWidth()-28, 17, p2);
}
```

Como puede comprobarse, a estos métodos se les pasará como parámetro las coordenadas del elemento a dibujar relativas al espacio ocupado por el control y el pincel a utilizar en cada caso.

Hecho esto, ya tenemos finalizado nuestro cuadro de texto personalizado con contador de caracteres. Para añadirlo a la interfaz de nuestra aplicación lo incluiremos en el layout XML de la ventana tal como haríamos con cualquier otro control, teniendo en cuenta que deberemos hacer referencia a él con el nombre completo de la nueva clase creada (incluido el paquete java), que en mi caso particular sería `net.sgoliver.ExtendedEditText`.

```
<net.sgoliver.ExtendedEditText
    android:id="@+id/TxtPrueba"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
```

En el siguiente apartado veremos cómo crear un control personalizado utilizando la segunda de las opciones expuestas, es decir, combinando varios controles ya existentes. Comentaremos además como añadir eventos y propiedades personalizadas a nuestro control y cómo hacer referencia a dichas propiedades desde su definición XML.

> El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-ExtendedEditText.zip

Controles personalizados: Combinar controles

Ya vimos cómo Android ofrece tres formas diferentes de crear controles personalizados para nuestras aplicaciones y dedicamos el capítulo anterior a comentar la primera de las posibilidades, que consistía en extender la funcionalidad de un control ya existente.

En este capítulo sobre el tema vamos a centrarnos en la creación de controles compuestos, es decir, controles personalizados construidos a partir de varios controles estándar, combinando la funcionalidad de todos ellos en un sólo control reutilizable en otras aplicaciones.

Como ejemplo ilustrativo vamos a crear un control de identificación (*login*) formado por varios controles estándar de Android. La idea es conseguir un control como el que se muestra la siguiente imagen esquemática:

A efectos didácticos, y para no complicar más el ejemplo, vamos a añadir también a la derecha del botón `Login` una etiqueta donde mostrar el resultado de la identificación del usuario (login correcto o incorrecto).

A este control añadiremos además eventos personalizados, veremos cómo añadirlo a nuestras aplicaciones, y haremos que se pueda personalizar su aspecto desde el layout XML de nuestra interfaz utilizando también atributos XML personalizados.

Empecemos por el principio. Lo primero que vamos a hacer es construir la interfaz de nuestro control a partir de controles sencillos: etiquetas, cuadros de texto y botones. Para ello vamos a crear un nuevo layout XML en la carpeta `\res\layout` con el nombre "`control_login.xml`".

En este fichero vamos a definir la estructura del control como ya hemos visto en muchos apartados anteriores, sin ninguna particularidad destacable. Para este caso quedaría como sigue:

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical"
  android:padding="10dip">

    <TextView android:id="@+id/TextView01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Usuario:"
        android:textStyle="bold" />

    <EditText android:id="@+id/TxtUsuario"
        android:layout_height="wrap_content"
        android:layout_width="fill_parent" />

    <TextView android:id="@+id/TextView02"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Contraseña:"
        android:textStyle="bold" />

    <EditText android:id="@+id/TxtPassword"
        android:layout_height="wrap_content"
        android:layout_width="fill_parent" />

    <LinearLayout android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >

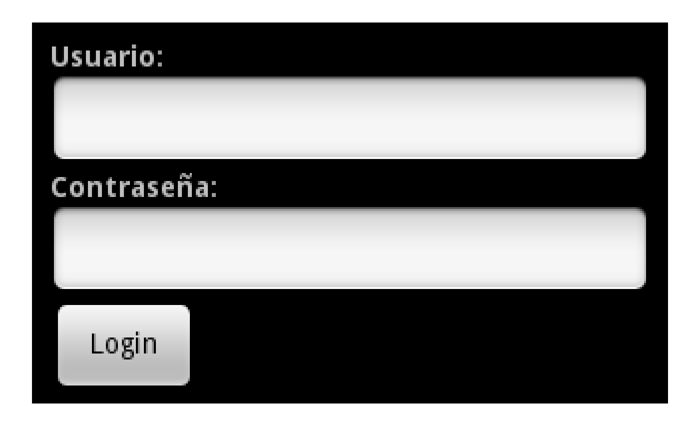
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/BtnAceptar"
            android:text="Login"
            android:paddingLeft="20dip"
            android:paddingRight="20dip" />

        <TextView android:id="@+id/LblMensaje"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:paddingLeft="10dip"
            android:textStyle="bold" />

    </LinearLayout>

</LinearLayout>
```

Visualmente, nuestro control ya quedaría como se observa en la siguiente imagen:



A continuación crearemos su clase java asociada donde definiremos toda la funcionalidad de nuestro control. Dado que nos hemos basado en un `LinearLayout` para construir el control, esta nueva clase deberá heredar también de la clase java `LinearLayout` de Android. Redefiniremos además los dos constructores básicos:

```
public class ControlLogin extends LinearLayout
{
     public ControlLogin(Context context) {
            super(context);
            inicializar();
     }

     public ControlLogin(Context context, AttributeSet attrs) {
    super(context, attrs);
    inicializar();
      }
}
```

Como se puede observar, todo el trabajo lo dejamos para el método `inicializar()`. En este método inflaremos el layout XML que hemos definido, obtendremos las referencias a todos los controles y asignaremos los eventos necesarios. Todo esto ya lo hemos hecho en otras ocasiones, por lo que tampoco nos vamos a detener mucho. Veamos cómo queda el método completo:

```
private void inicializar()
{
    //Utilizamos el layout 'control_login' como interfaz del control
    String infService = Context.LAYOUT_INFLATER_SERVICE;
    LayoutInflater li =
        (LayoutInflater)getContext().getSystemService(infService);
    li.inflate(R.layout.control_login, this, true);

    //Obtenemoslas referencias a los distintos control
    txtUsuario = (EditText)findViewById(R.id.TxtUsuario);
    txtPassword = (EditText)findViewById(R.id.TxtPassword);
    btnLogin = (Button)findViewById(R.id.BtnAceptar);
    lblMensaje = (TextView)findViewById(R.id.LblMensaje);

    //Asociamos los eventos necesarios
    asignarEventos();
}
```

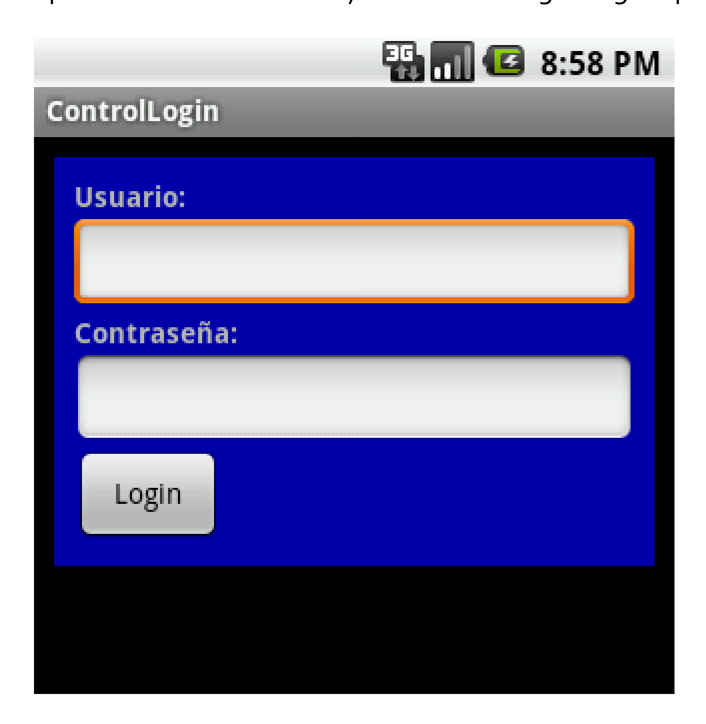
Dejaremos por ahora a un lado el método `asignarEventos()`, volveremos sobre él más tarde.

Con esto ya tenemos definida la interfaz y la funcionalidad básica del nuevo control por lo que ya podemos utilizarlo desde otra actividad como si se tratase de cualquier otro control predefinido. Para ello haremos referencia a él utilizando la ruta completa del paquete java utilizado, en nuestro caso quedaría como `net.sgoliver.ControlLogin`. Vamos a insertar nuestro nuevo control en la actividad principal de la aplicación:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10dip" >

    <net.sgoliver.ControlLogin android:id="@+id/CtlLogin"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="#0000AA" />
</LinearLayout>
```

Dado que estamos heredando de un `LinearLayout` podemos utilizar en principio cualquier atributo permitido para dicho tipo de controles, en este caso hemos establecido por ejemplo los atributos `layout_width`, `layout_height` y `background`. Si ejecutamos ahora la aplicación veremos cómo ya hemos conseguido gran parte de nuestro objetivo:



Vamos a añadir ahora algo más de funcionalidad. En primer lugar, podemos añadir algún método público exclusivo de nuestro control. Como ejemplo podemos añadir un método que permita modificar el texto de la etiqueta de resultado del login.

```
public void setMensaje(String msg)
{
    lblMensaje.setText(msg);
}
```

En segundo lugar, todo control que se precie debe tener algunos eventos que nos permitan responder a las acciones del usuario de la aplicación. Así por ejemplo, los botones tienen un evento `OnClick`, las listas un evento `OnItemSelected`, etc. Pues bien, nosotros vamos a dotar a nuestro control de un evento personalizado, llamado `OnLogin`, que se lance cuando el usuario introduce sus credenciales de identificación y pulsa el botón "*Login*".

Para ello, lo primero que vamos a hacer es concretar los detalles de dicho evento, creando una interfaz java para definir su *listener*. Esta interfaz tan sólo tendrá un método llamado `onLogin()` que devolverá los dos datos introducidos por el usuario (usuario y contraseña).

```
package net.sgoliver;

public interface OnLoginListener
{
    void onLogin(String usuario, String password);
}
```

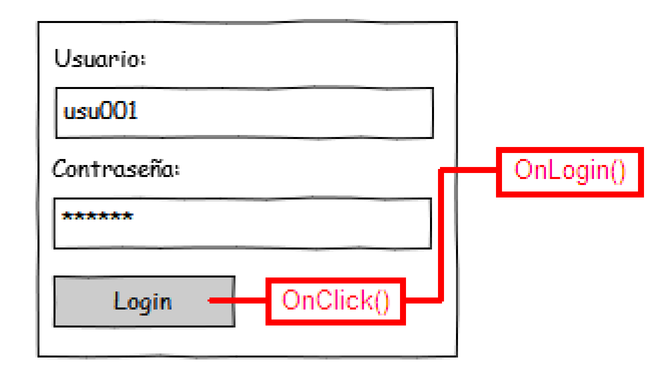
A continuación, deberemos añadir un nuevo miembro de tipo `OnLoginListener` a la clase `ControlLogin`, y un método público que permita suscribirse al nuevo evento.

```
public class ControlLogin extends LinearLayout
{
     private OnLoginListener listener;

     //...

     public void setOnLoginListener(OnLoginListener l)
     {
    listener = l;
     }
}
```

Con esto, la aplicación principal ya puede suscribirse al evento `OnLogin` y ejecutar su propio código cuando éste se genere. ¿Pero cuándo se genera exactamente? Dijimos antes que queremos lanzar el evento `OnLogin` cuando el usuario pulse el botón "*Login*" de nuestro control. Pues bien, para hacerlo, volvamos al método `asignarEventos()` que antes dejamos aparcado. En este método vamos a implementar el evento `OnClick` del botón de *Login* para lanzar el nuevo evento `OnLogin` del control. ¿Confundido?. Intento explicarlo de otra forma. Vamos a aprovechar el evento `OnClick()` del botón *Login* (que es un evento interno a nuestro control, no se verá desde fuera) para lanzar hacia el exterior el evento `OnLogin()` (que será el que debe capturar y tratar la aplicación que haga uso del control).



Para ello, implementaremos el evento `OnClick` como ya hemos hecho en otras ocasiones y como acciones generaremos el evento `OnLogin` de nuestro *listener* pasándole los datos que ha introducido el usuario en los cuadros de texto "*Usuario*" y "*Contraseña*":

```
private void asignarEventos()
{
    btnLogin.setOnClickListener(new OnClickListener()
    {
        @Override
        public void onClick(View v) {
            listener.onLogin(txtUsuario.getText().toString(),
                         txtPassword.getText().toString());
        }
    });
}
```

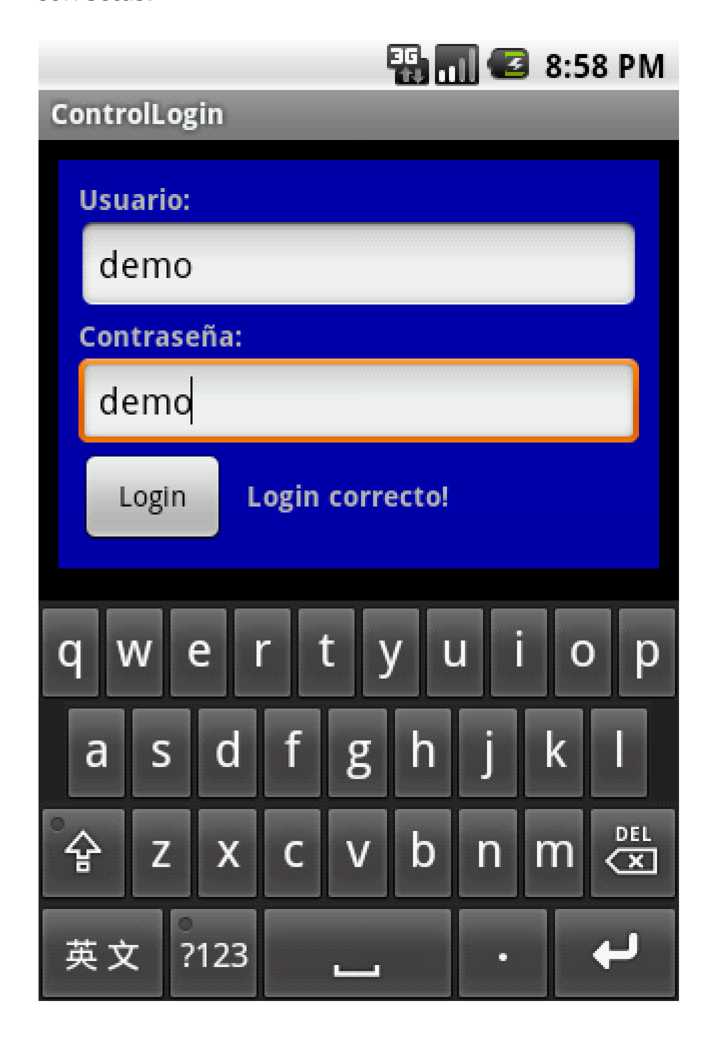
Con todo esto, la aplicación principal ya puede implementar el evento `OnLogin` de nuestro control, haciendo por ejemplo la validación de las credenciales del usuario y modificando convenientemente el texto de la etiqueta de resultado:

```
@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ctlLogin = (ControlLogin)findViewById(R.id.CtlLogin);

    ctlLogin.setOnLoginListener(new OnLoginListener()
    {
    @Override
    public void onLogin(String usuario, String password)
    {
        //Validamos el usuario y la contraseña
        if (usuario.equals("demo") && password.equals("demo"))
            ctlLogin.setMensaje("Login correcto!");
        else
            ctlLogin.setMensaje("Vuelva a intentarlo.");
    }
    });
}
```

Veamos lo que ocurre al ejecutar ahora la aplicación principal e introducir las credenciales correctas:



Nuestro control está ya completo, en aspecto y funcionalidad. Hemos personalizado su interfaz y hemos añadido métodos y eventos propios. ¿Podemos hacer algo más? Pues sí.

Cuando vimos cómo añadir el control de login al layout de la aplicación principal dijimos que podíamos utilizar cualquier atributo xml permitido para el contenedor `LinearLayout`, ya que nuestro control derivaba de éste. Pero vamos a ir más allá y vamos a definir también atributos xml exclusivos para nuestro control. Como ejemplo, vamos a definir un atributo llamado `login_text` que permita establecer el texto del botón de Login desde el propio layout xml, es decir, en tiempo de diseño.

Primero vamos de declarar el nuevo atributo y lo vamos a asociar al control `ControlLogin`. Esto debe hacerse en el fichero `\res\values\attrs.xml`. Para ello, añadiremos una nueva sección `<declare-styleable>` asociada a `ControlLogin` dentro del elemento `<resources>`, donde indicaremos el nombre (`name`) y el tipo (`format`) del nuevo atributo.

```
<resources>
    <declare-styleable name="ControlLogin">
        <attr name="login_text" format="string"/>
    </declare-styleable>
</resources>
```

En nuestro caso, el tipo del atributo será `string`, dado que contendrá una cadena de texto con el mensaje a mostrar en el botón.

Con esto ya tendremos permitido el uso del nuevo atributo dentro del layout de la aplicación principal. Ahora nos falta procesar el atributo desde nuestro control personalizado. Este tratamiento lo podemos hacer en el constructor de la clase `ControlLogin`. Para ello, obtendremos la lista de atributos asociados a `ControlLogin` mediante el método `obtainStyledAttributes()` del contexto de la aplicación, obtendremos el valor del nuevo atributo definido (mediante su ID, que estará formado por la concatenación del nombre del control y el nombre del atributo, en nuestro caso `ControlLogin_login_text`) y modificaremos el texto por defecto del control con el nuevo texto.

```
public ControlLogin(Context context, AttributeSet attrs) {
    super(context, attrs);
    inicializar();

    // Procesamos los atributos XML personalizados
    TypedArray a =
        getContext().obtainStyledAttributes(attrs,
            R.styleable.ControlLogin);

    String textoBoton = a.getString(
            R.styleable.ControlLogin_login_text);

    btnLogin.setText(textoBoton);

    a.recycle();
}
```

Ya sólo nos queda utilizarlo. Para ello debemos primero declarar un nuevo espacio de nombres (*namespace*) local para el paquete java utilizado, que en nuestro caso he llamado "`sgo`":

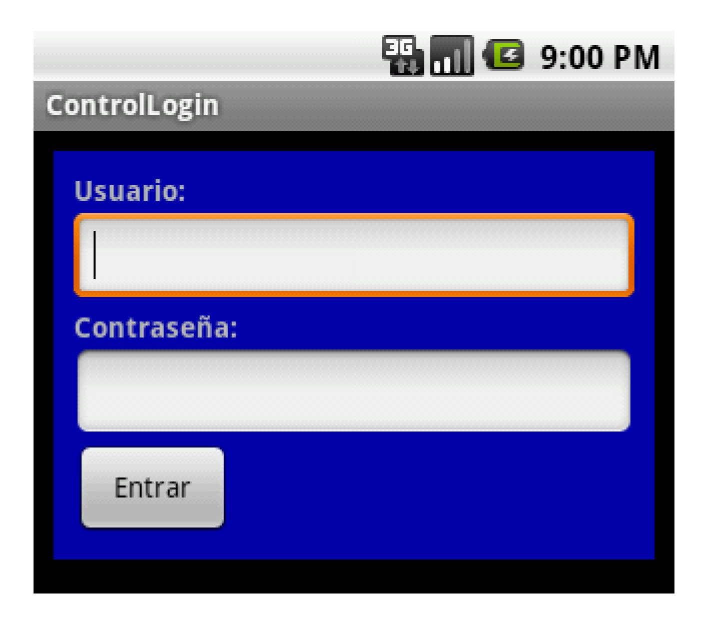
```
xmlns:sgo="http://schemas.android.com/apk/res/net.sgoliver"
```

Tras esto, sólo queda asignar el valor del nuevo atributo precedido del nuevo namespace, por ejemplo con el texto "Entrar":

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:sgo="http://schemas.android.com/apk/res/net.sgoliver"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10dip" >

    <net.sgoliver.ControlLogin android:id="@+id/CtlLogin"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="#0000AA"
        sgo:login_text="Entrar" />
</LinearLayout>
```

Finalmente, si ejecutamos de nuevo la aplicación principal veremos cómo el botón de login se inicializa con el texto definido en el atributo `login_text` y que todo continúa funcionando correctamente.



Como resumen, en este apartado hemos visto cómo construir un control Android personalizado a partir de otros controles estándar, componiendo su interfaz, añadiendo métodos y eventos personalizados, e incluso añadiendo nuevas opciones en tiempo de diseño añadiendo atributos XML exclusivos.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-controllogin.zip

Controles personalizados: Diseño completo

En apartados anteriores del curso ya comentamos dos de las posibles vías que tenemos para crear controles personalizados en Android: la primera de ellas extendiendo la funcionalidad de un control ya existente, y como segunda opción creando un nuevo control compuesto por otros más sencillos.

En este nuevo apartado vamos a describir la tercera de las posibilidades que teníamos disponibles, que consiste en crear un control completamente desde cero, sin utilizar como base otros controles existentes. Como ejemplo, vamos a construir un control que nos permita seleccionar un color entre varios disponibles.

Los colores disponibles van a ser sólo cuatro, que se mostrarán en la franja superior del control. En la parte inferior se mostrará el color seleccionado en cada momento, o permanecerá negro si aún no se ha seleccionado ningún color. Valga la siguiente imagen como muestra del aspecto que tendrá nuestro control de selección de color:



Por supuesto este control no tiene mucha utilidad práctica dada su sencillez, pero creo que puede servir como ejemplo para comentar los pasos necesarios para construir cualquier otro control más complejo. Empecemos.

En las anteriores ocasiones vimos cómo el nuevo control creado siempre heredaba de algún otro control o contenedor ya existente. En este caso sin embargo, vamos a heredar nuestro control directamente de la clase `View` (clase padre de la gran mayoría de elementos visuales de Android). Esto implica, entre otras cosas, que por defecto nuestro control no va a tener ningún tipo de interfaz gráfica, por lo que todo el trabajo de "dibujar" la interfaz lo vamos a tener que hacer nosotros. Además, como paso previo a la representación gráfica de la interfaz, también vamos a tener que determinar las dimensiones que nuestro control tendrá dentro de su elemento contenedor. Como veremos ahora, ambas cosas se llevarán a cabo redefiniendo dos eventos de la clase View, `onDraw()` para el dibujo de la interfaz, y `onMeasure()` para el cálculo de las dimensiones.

Por llevar un orden cronológico, empecemos comentando el evento `onMeasure()`. Este evento se ejecuta automáticamente cada vez que se necesita recalcular el tamaño de un control. Pero como ya hemos visto en varias ocasiones, los elementos gráficos incluidos en una aplicación Android se distribuyen por la pantalla de una forma u otra dependiendo del tipo de contenedor o layout utilizado. Por tanto, el tamaño de un control determinado en la pantalla no dependerá sólo de él, sino de ciertas restricciones impuestas por su elemento contenedor o elemento padre. Para resolver esto, en el evento `onMeasure()` recibiremos como parámetros las restricciones del elemento padre en cuanto a ancho y alto del control, con lo que podremos tenerlas en cuenta a la hora de determinar el ancho y alto de nuestro control personalizado. Estas restricciones se reciben en forma de objetos `MeasureSpec`, que contiene dos campos: modo y tamaño. El significado del segundo de ellos es obvio, el primero por su parte sirve para matizar el significado del segundo. Me explico. Este campo modo puede contener tres valores posibles:

Valor	**Descripción**
`AT_MOST`	Indica que el control podrá tener como máximo el tamaño especificado.
`EXACTLY`	Indica que al control se le dará exactamente el tamaño especificado.
`UNSPECIFIED`	Indica que el control padre no impone ninguna restricción sobre el tamaño.

Dependiendo de esta pareja de datos, podremos calcular el tamaño deseado para nuestro control. Para nuestro control de ejemplo, apuraremos siempre el tamaño máximo disponible (o un tamaño por defecto de 200*100 en caso de no recibir ninguna restricción), por lo que en todos los casos elegiremos como tamaño de nuestro control el tamaño recibido como parámetro:

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec)
{
    int ancho = calcularAncho(widthMeasureSpec);
    int alto = calcularAlto(heightMeasureSpec);

    setMeasuredDimension(ancho, alto);
}

private int calcularAlto(int limitesSpec)
{
    int res = 100; //Alto por defecto

    int modo = MeasureSpec.getMode(limitesSpec);
    int limite = MeasureSpec.getSize(limitesSpec);

    if (modo == MeasureSpec.AT_MOST) {
        res = limite;
    }
    else if (modo == MeasureSpec.EXACTLY) {
        res = limite;
    }

    return res;
}

private int calcularAncho(int limitesSpec)
{
    int res = 200; //Ancho por defecto

    int modo = MeasureSpec.getMode(limitesSpec);
    int limite = MeasureSpec.getSize(limitesSpec);

    if (modo == MeasureSpec.AT_MOST) {
        res = limite;
    }
    else if (modo == MeasureSpec.EXACTLY) {
        res = limite;
    }

    return res;
}
```

Como nota importante, al final del evento `onMeasure()` siempre debemos llamar al método `setMeasuredDimension()` pasando como parámetros el ancho y alto calculados para nuestro control.

Con esto ya hemos determinado las dimensiones del control, por lo que tan sólo nos queda dibujar su interfaz gráfica. Como hemos indicado antes, esta tarea se realiza dentro del evento `onDraw()`. Este evento recibe como parámetro un objeto de tipo `Canvas`, sobre el que podremos ejecutar todas las operaciones de dibujo de la interfaz. No voy a entrar en detalles de la clase `Canvas`, porque ése será tema central de un próximo apartado. Por ahora nos vamos a conformar sabiendo que es la clase que contiene la mayor parte de los métodos

de dibujo en interfaces Android, por ejemplo drawRect() para dibujar rectángulos, drawCircle() para círculos, drawBitmap() para imagenes, drawText() para texto, e infinidad de posibilidades más. Para consultar todos los métodos disponibles puedes dirigirte a la documentación oficial de la clase Canvas de Android. Además de la clase Canvas, también me gustaría destacar la clase Paint, que permite definir el estilo de dibujo a utilizar en los métodos de dibujo de Canvas, por ejemplo el ancho de trazado de las líneas, los colores de relleno, etc.

Para nuestro ejemplo no necesitaríamos conocer nada más, ya que la interfaz del control es relativamente sencilla. Vemos primero el código y después comentamos los pasos realizados:

```
@Override
protected void onDraw(Canvas canvas)
{
    //Obtenemos las dimensiones del control
    int alto = getMeasuredHeight();
    int ancho = getMeasuredWidth();

    //Colores Disponibles
    Paint pRelleno = new Paint();
    pRelleno.setStyle(Style.FILL);

    pRelleno.setColor(Color.RED);
    canvas.drawRect(0, 0, ancho/4, alto/2, pRelleno);

    pRelleno.setColor(Color.GREEN);
    canvas.drawRect(ancho/4, 0, 2*(ancho/4), alto/2, pRelleno);

    pRelleno.setColor(Color.BLUE);
    canvas.drawRect(2*(ancho/4), 0, 3*(ancho/4), alto/2, pRelleno);

    pRelleno.setColor(Color.YELLOW);
    canvas.drawRect(3*(ancho/4), 0, 4*(ancho/4), alto/2, pRelleno);

    //Color Seleccionado
    pRelleno.setColor(colorSeleccionado);
    canvas.drawRect(0, alto/2, ancho, alto, pRelleno);

    //Marco del control
    Paint pBorde = new Paint();
    pBorde.setStyle(Style.STROKE);
    pBorde.setColor(Color.WHITE);
    canvas.drawRect(0, 0, ancho-1, alto/2, pBorde);
    canvas.drawRect(0, 0, ancho-1, alto-1, pBorde);
}
```

En primer lugar obtenemos las dimensiones calculadas en la última llamada a onMeasure() mediante los métodos getMeasuredHeight() y getMeasuredWidth(). Posteriormente definimos un objeto Paint que usaremos para dibujar los rellenos de cada color seleccionable. Para indicar que se trata del color de relleno a utilizar utilizaremos la llamada a setStyle(Style.FILL). Tras esto, ya sólo debemos dibujar cada uno de los cuadros en su posición correspondiente con drawRect(), estableciendo antes de cada uno de ellos el color deseado con setColor(). Por último, dibujamos el marco del control definiendo un nuevo objeto Paint, esta vez con estilo Style.STROKE dado que se utilizará para dibujar sólo líneas, no rellenos.

Con esto, ya deberíamos tener un control con el aspecto exacto que definimos en un principio. El siguiente paso será definir su funcionalidad implementando los eventos a los que queramos que responda nuestro control, tanto eventos internos como externos.

En nuestro caso sólo vamos a tener un evento de cada tipo. En primer lugar definiremos un evento interno (evento que sólo queremos capturar de forma interna al control, sin exponerlo al usuario) para responder a las pulsaciones del usuario sobre los colores de la zona superior, y que utilizaremos para actualizar el color de la zona inferior con el color seleccionado. Para ello implementaremos el evento `onTouch()`, lanzado cada vez que el usuario toca la pantalla sobre nuestro control. La lógica será sencilla, simplemente consultaremos las coordenadas donde ha pulsado el usuario (mediante los métodos `getX()` y `getY()`), y dependiendo del lugar pulsado determinaremos el color sobre el que se ha seleccionado y actualizaremos el valor del atributo `colorSeleccionado`. Finalmente, llamamos al método `invalidate()` para refrescar la interfaz del control, reflejando así el cambio en el color seleccionado, si se ha producido.

```
@Override
public boolean onTouchEvent(MotionEvent event)
{
    //Si se ha pulsado en la zona superior
    if (event.getY() > 0 && event.getY() < (getMeasuredHeight()/2))
    {
        //Si se ha pulsado dentro de los límites del control
        if (event.getX() > 0 && event.getX() < getMeasuredWidth())
        {
            //Determinamos el color seleccionado según el punto pulsado
            if(event.getX() > (getMeasuredWidth()/4)*3)
                colorSeleccionado = Color.YELLOW;
            else if(event.getX() > (getMeasuredWidth()/4)*2)
                colorSeleccionado = Color.BLUE;
            else if(event.getX() > (getMeasuredWidth()/4)*1)
                colorSeleccionado = Color.GREEN;
            else
                colorSeleccionado = Color.RED;

            //Refrescamos el control
            this.invalidate();
        }
    }

    return super.onTouchEvent(event);
}
```

En segundo lugar crearemos un evento externo personalizado, que lanzaremos cuando el usuario pulse sobre la zona inferior del control, como una forma de aceptar definitivamente el color seleccionado. Llamaremos a este evento `onSelectedColor()`. Para crearlo actuaremos de la misma forma que ya vimos en el capítulo anterior. Primero definiremos una interfaz para el listener de nuestro evento:

```
package net.sgoliver.android;

public interface OnColorSelectedListener
{
    void onColorSelected(int color);
}
```

Posteriormente, definiremos un objeto de este tipo como atributo de nuestro control y escribiremos un nuevo método que permita a las aplicaciones suscribirse al evento:

```
public class SelectorColor extends View
{
     private OnColorSelectedListener listener;

     //...

     public void setOnColorSelectedListener(OnColorSelectedListener l)
     {
         listener = l;
     }
}
```

Y ya sólo nos quedaría lanzar el evento en el momento preciso. Esto también lo haremos dentro del evento `onTouch()`, cuando detectemos que el usuario ha pulsado en la zona inferior de nuestro control:

```
@Override
public boolean onTouchEvent(MotionEvent event)
{
    //Si se ha pulsado en la zona superior
    if (event.getY() > 0 && event.getY() < (getMeasuredHeight()/2))
    {
                //...
    }
    //Si se ha pulsado en la zona inferior
    else if (event.getY() > (getMeasuredHeight()/2) &&
                 event.getY() < getMeasuredHeight())
    {
        //Lanzamos el evento externo de selección de color
        listener.onColorSelected(colorSeleccionado);
    }

    return super.onTouchEvent(event);
}
```

Con esto, nuestra aplicación principal ya podría suscribirse a este nuevo evento para estar informada cada vez que se seleccione un color. Sirva la siguiente plantilla a modo de ejemplo:

```
public class ControlPersonalizado extends Activity
{
    private SelectorColor ctlColor;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        ctlColor = (SelectorColor)findViewById(R.id.scColor);

        ctlColor.setOnColorSelectedListener(new OnColorSelectedListener()
        {
            @Override
            public void onColorSelected(int color)
            {
                //Aquí se trataría el color seleccionado (parámetro 'color'
                //...
            }
        });
```

```
    }
}
```

Con esto, tendríamos finalizado nuestro control completamente personalizado, que hemos construido sin utilizar como base ningún otro control predefinido, definiendo desde cero tanto su aspecto visual como su funcionalidad interna o sus eventos públicos.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-selector-color.zip

3

Widgets de Escritorio

III. Widgets

Widgets básicos

En los dos próximos capítulos de este Curso de Programación Android vamos a describir cómo crear un *widget* de escritorio (*home screen widget*).

En esta primera parte construiremos un *widget* estático (no será interactivo, ni contendrá datos actualizables, ni responderá a eventos) muy básico para entender claramente la estructura interna de un componente de este tipo, y en el siguiente capítulo completaremos el ejercicio añadiendo una ventana de configuración inicial para el widget, añadiremos algún dato que podamos actualizar periódicamente, y haremos que responda a pulsaciones del usuario.

Como hemos dicho, en esta primera parte vamos a crear un *widget* muy básico, consistente en un simple marco rectangular negro con un mensaje de texto predeterminado ("*Mi Primer Widget*"). La sencillez del ejemplo nos permitirá centrarnos en los pasos principales de la construcción de un widget Android y olvidarnos de otros detalles que nada tienen que ver con el tema que nos ocupa (gráficos, datos, ...). Para que os hagáis una idea, éste será el aspecto final de nuestro widget de ejemplo:



Los pasos principales para la creación de un widget Android son los siguientes:

1. Definición de su interfaz gráfica (*layout*).
2. Configuración XML del widget (AppWidgetProviderInfo).
3. Implementación de la funcionalidad del widget (AppWidgetProvider) , especialmente su evento de actualización.
4. Declaración del widget en el *Android Manifest* de la aplicación.

En el primer paso no nos vamos a detener mucho ya que es análogo a cualquier definición de layout de las que hemos visto hasta ahora en el curso. En esta ocasión, la interfaz del widget estará compuesta únicamente por un par de *frames* (FrameLayout), uno negro exterior y uno blanco interior algo más pequeño para simular el marco, y una etiqueta de texto (TextView) que albergará el mensaje a mostrar. Veamos cómo queda el layout xml, que para este ejemplo llamaremos "miwidget.xml":

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="5dip">

    <FrameLayout
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:background="#000000"
        android:padding="10dip" >

        <FrameLayout
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:background="#FFFFFF"
            android:padding="5dip" >

            <TextView android:id="@+id/txtMensaje"
                android:layout_width="fill_parent"
                android:layout_height="fill_parent"
                android:textColor="#000000"
                android:text="Mi Primer Widget" />

        </FrameLayout>

    </FrameLayout>

</LinearLayout>
```

Cabe destacar aquí que, debido a que el layout de los widgets de Android está basado en un tipo especial de componentes llamados RemoteViews, no es posible utilizar en su interfaz todos los contenedores y controles que hemos visto en apartados anteriores sino unos pocos básicos que se indican a continuación:

- Contenedores: FrameLayout, LinearLayout y RelativeLayout
- Controles: Button, ImageButton, ImageView, TextView, ProgressBar, Chronometer y AnalogClock.

Aunque la lista de controles soportados no deja de ser curiosa (al menos en mi humilde opinión), debería ser suficiente para crear todo tipo de widgets.

Como segundo paso del proceso de construcción vamos a crear un nuevo fichero XML donde definiremos un conjunto de propiedades del widget, como por ejemplo su tamaño en pantalla o su frecuencia de actualización. Este XML se deberá crear en la carpeta \res\xml de nuestro proyecto.

En nuestro caso de ejemplo lo llamaremos "miwidget_wprovider.xml" y tendrá la siguiente estructura:

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:initialLayout="@layout/miwidget"
    android:minWidth="146dip"
    android:minHeight="72dip"
    android:label="Mi Primer Widget"
    android:updatePeriodMillis="3600000"
/>
```

Para nuestro widget estamos definiendo las siguientes propiedades:

- initialLayout: referencia al layout XML que hemos creado en el paso anterior.
- minWidth: ancho mínimo del widget en pantalla, en dp (*density-independent pixels*).
- minHeight: alto mínimo del widget en pantalla, en dp (*density-independent pixels*).
- label: nombre del widget que se mostrará en el menú de selección de Android.
- updatePeriodMillis: frecuencia de actualización del widget, en milisegundos.

Existen varias propiedades más que se pueden definir. En el siguiente apartado utilizaremos alguna de ellas, el resto se pueden consultar en la documentación oficial de la clase AppWidgetProviderInfo.

Como sabemos, la pantalla inicial de Android se divide en 4×4 celdas donde se pueden colocar aplicaciones, accesos directos y widgets. Teniendo en cuenta las diferentes dimensiones de estas celdas según la orientación de la pantalla, existe una fórmula sencilla para ajustar las dimensiones de nuestro widget para que ocupe un número determinado de celdas sea cual sea la orientación:

- ancho_mínimo = (num_celdas * 74) – 2
- alto_mínimo = (num_celdas * 74) – 2

Atendiendo a esta fórmula, si queremos que nuestro widget ocupe un tamaño mínimo de 2 celdas de ancho por 1 celda de alto, deberemos indicar unas dimensiones de 146dp x 72dp.

Vamos ahora con el tercer paso. Éste consiste en implementar la funcionalidad de nuestro widget en su clase java asociada. Esta clase deberá heredar de AppWidgetProvider, que a su vez no es más que una clase auxiliar derivada de BroadcastReceiver, ya que los widgets de Android no son más que un caso particular de este tipo de componentes.

En esta clase deberemos implementar los mensajes a los que vamos a responder desde nuestro widget, entre los que destacan:

- `onEnabled()`: lanzado cuando se añade al escritorio la primera instancia de un widget.
- `onUpdate()`: lanzado periódicamente cada vez que se debe actualizar un widget.
- `onDeleted()`: lanzado cuando se elimina del escritorio una instancia de un widget.
- `onDisabled()`: lanzado cuando se elimina del escritorio la última instancia de un widget.

En la mayoría de los casos, tendremos que implementar como mínimo el evento `onUpdate()`. El resto de métodos dependerán de la funcionalidad de nuestro widget. En nuestro caso particular no nos hará falta ninguno de ellos ya que el widget que estamos creando no contiene ningún dato actualizable, por lo que crearemos la clase, llamada `MiWidget`, pero dejaremos vacío por el momento el método `onUpdate()`. En el siguiente apartado veremos qué cosas podemos hacer dentro de estos métodos.

```
package net.sgoliver.android;

import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.Context;

public class MiWidget extends AppWidgetProvider {
    @Override
    public void onUpdate(Context context,
                 AppWidgetManager appWidgetManager,
                 int[] appWidgetIds) {
        //Actualizar el widget
        //...
    }
}
```

El último paso del proceso será declarar el widget dentro del *manifest* de nuestra aplicación. Para ello, editaremos el fichero AndroidManifest.xml para incluir la siguiente declaración dentro del elemento `<application>`:

```
<application>
        ...
    <receiver android:name=".MiWidget" android:label="Mi Primer Widget">
        <intent-filter>
           <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
        </intent-filter>
        <meta-data
           android:name="android.appwidget.provider"
           android:resource="@xml/miwidget_wprovider" />
    </receiver>
</application>
```

El widget se declarará como un elemento `<receiver>` y deberemos aportar la siguiente información:

- Atributo `name`: Referencia a la clase java de nuestro widget, creada en el paso anterior.
- Elemento `<intent-filter>`, donde indicaremos los "eventos" a los que responderá nuestro widget, normalmente añadiremos el evento `APPWIDGET_UPDATE`, para detectar la acción de actualización.
- Elemento `<meta-data>`, donde haremos referencia con su atributo `resource` al XML de configuración que creamos en el segundo paso del proceso.

Con esto habríamos terminado de escribir los distintos elementos necesarios para hacer funcionar nuestro widget básico de ejemplo. Para probarlo, podemos ejecutar el proyecto de Eclipse en el emulador de Android, esperar a que se ejecute la aplicación principal (que estará vacía, ya que no hemos incluido ninguna funcionalidad para ella), ir a la pantalla principal del emulador y añadir nuestro widget al escritorio tal cómo lo haríamos en nuestro teléfono (pulsación larga sobre el escritorio o tecla Menú, seleccionar la opción Widgets, y por último seleccionar nuestro Widget). Os dejo una demostración en video (Nota: se trata de un enlace a *YouTube*, se necesita conexión a internet para poder visualizarlo).

Como podéis ver en el video, ya hemos conseguido la funcionalidad básica de un widget, es posible añadir varias instancias al escritorio, desplazarlos por la pantalla y eliminarlos enviándolos a la papelera.

En el próximo apartado veremos cómo podemos mejorar este widget añadiendo una pantalla de configuración inicial, mostraremos algún dato que se actualice periódicamente, y añadiremos la posibilidad de capturar eventos de pulsación sobre el widget.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-widgets-1.zip

Widgets avanzados

Ya hemos visto cómo construir un widget básico para Android, y prometimos que dedicaríamos un apartado adicional a comentar algunas características más avanzadas de este tipo de componentes. Pues bien, en este segundo apartado sobre el tema vamos a ver cómo podemos añadir los siguientes elementos y funcionalidades al widget básico que ya construimos:

- Pantalla de configuración inicial.
- Datos actualizables de forma periódica.
- Eventos de usuario.

Como sabéis, intento simplificar al máximo todos los ejemplos que utilizo en este curso para que podamos centrar nuestra atención en los aspectos realmente importantes. En esta ocasión utilizaré el mismo criterio y las únicas características (aunque suficientes para

demostrar los tres conceptos anteriores) que añadiremos a nuestro widget serán las siguientes:

1. Añadiremos una pantalla de configuración inicial del widget, que aparecerá cada vez que se añada una nueva instancia del widget a nuestro escritorio. En esta pantalla podrá configurarse únicamente el mensaje de texto a mostrar en el widget.
2. Añadiremos un nuevo elemento de texto al widget que muestre la hora actual. Esto nos servirá para comprobar que el widget se actualiza periódicamente.
3. Añadiremos un botón al widget, que al ser pulsado forzará la actualización inmediata del mismo.

Empecemos por el primer punto, la pantalla de configuración inicial del widget. Y procederemos igual que para el diseño de cualquier otra actividad Android, definiendo su layout xml. En nuestro caso será muy sencilla, un cuadro de texto para introducir el mensaje a personalizar y dos botones, uno para aceptar la configuración y otro para cancelar (en cuyo caso el widget no se añade al escritorio). En esta ocasión llamaremos a este layout "`widget_config.xml`". Veamos cómo queda:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical">

    <TextView android:id="@+id/LblMensaje"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Mensaje personalizado:" />

    <EditText android:id="@+id/TxtMensaje"
              android:layout_height="wrap_content"
              android:layout_width="fill_parent" />

    <LinearLayout
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:orientation="horizontal" >

        <Button android:id="@+id/BtnAceptar"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Aceptar" />

        <Button android:id="@+id/BtnCancelar"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Cancelar" />

    </LinearLayout>

</LinearLayout>
```

Una vez diseñada la interfaz de nuestra actividad de configuración tendremos que implementar su funcionalidad en java. Llamaremos a la clase `WidgetConfig`, su estructura será análoga a la de cualquier actividad de Android, y las acciones a realizar serán las comentadas a continuación. En primer lugar nos hará falta el identificador de la instancia

concreta del widget que se configurará con esta actividad. Este ID nos llega como parámetro del *intent* que ha lanzado la actividad. Como ya vimos en un apartado anterior del curso, este *intent* se puede recuperar mediante el método `getIntent()` y sus parámetros mediante el método `getExtras()`. Conseguida la lista de parámetros del intent, obtendremos el valor del ID del widget accediendo a la clave `AppWidgetManager.EXTRA_APPWIDGET_ID`. Veamos el código hasta este momento:

```
public class WidgetConfig extends Activity {

    private int widgetId = 0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.widget_config);

        //Obtenemos el Intent que ha lanzado esta ventana
        //y recuperamos sus parámetros
        Intent intentOrigen = getIntent();
        Bundle params = intentOrigen.getExtras();

        //Obtenemos el ID del widget que se está configurando
        widgetId = params.getInt(
                AppWidgetManager.EXTRA_APPWIDGET_ID,
                AppWidgetManager.INVALID_APPWIDGET_ID);

        //Establecemos el resultado por defecto (si se pulsa el botón 'Atrás'
        //del teléfono será éste el resultado devuelto).
        setResult(RESULT_CANCELED);

        //...
    }
}
```

En el código también podemos ver como aprovechamos este momento para establecer el resultado por defecto a devolver por la actividad de configuración mediante el método `setResult()`. Esto es importante porque las actividades de configuración de widgets deben devolver siempre un resultado (`RESULT_OK` en caso de aceptarse la configuración, o `RESULT_CANCELED` en caso de salir de la configuración sin aceptar los cambios). Estableciendo aquí ya un resultado `RESULT_CANCELED` por defecto nos aseguramos de que si el usuario sale de la configuración pulsando el botón *Atrás* del teléfono no añadiremos el widget al escritorio, mismo resultado que si pulsáramos el botón "*Cancelar*" de nuestra actividad.

Como siguiente paso recuperamos las referencias a cada uno de los controles de la actividad de configuración:

```
//Obtenemos la referencia a los controles de la pantalla
final Button btnAceptar = (Button)findViewById(R.id.BtnAceptar);
final Button btnCancelar = (Button)findViewById(R.id.BtnCancelar);
final EditText txtMensaje = (EditText)findViewById(R.id.TxtMensaje);
```

Por último, implementaremos las acciones de los botones "*Aceptar*" y "*Cancelar*". En principio, el botón Cancelar no tendría por qué hacer nada, tan sólo finalizar la actividad

mediante una llamada al método `finish()` ya que el resultado `CANCELED` ya se ha establecido por defecto anteriormente:

```
//Implementación del botón "Cancelar"
btnCancelar.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        //Devolvemos como resultado: CANCELAR (RESULT_CANCELED)
        finish();
    }
});
```

En el caso del botón Aceptar tendremos que hacer más cosas:

1. Guardar de alguna forma el mensaje que ha introducido el usuario.
2. Actualizar manualmente la interfaz del widget según la configuración establecida.
3. Devolver el resultado `RESULT_OK` aportando además el ID del widget.

Para el primer punto nos ayudaremos de la API de Preferencias que describimos en el apartado anterior. En nuestro caso, guardaremos una sola preferencia cuya clave seguirá el patrón "`msg_IdWidget`", esto nos permitirá distinguir el mensaje configurado para cada instancia del widget que añadamos a nuestro escritorio de Android.

El segundo paso indicado es necesario debido a que si definimos una actividad de configuración para un widget, será ésta la que tenga la responsabilidad de realizar la primera actualización del mismo en caso de ser necesario. Es decir, tras salir de la actividad de configuración no se lanzará automáticamente el evento `onUpdate()` del widget (sí se lanzará posteriormente y de forma periódica según la configuración del parámetro `updatePeriodMillis` del provider que veremos más adelante), sino que tendrá que ser la propia actividad quien fuerce la primera actualización. Para ello, simplemente obtendremos una referencia al widget manager de nuestro contexto mediente el método `AppWidgetManager.getInstance()` y con esta referencia llamaremos al método estático de actualización del widget `MiWidget.actualizarWidget()`, que actualizará los datos de todos los controles del widget (lo veremos un poco más adelante).

Por último, al resultado a devolver (`RESULT_OK`) deberemos añadir información sobre el ID de nuestro widget. Esto lo conseguimos creando un nuevo Intent que contenga como parámetro el ID del widget que recuperamos antes y estableciéndolo como resultado de la actividad mediante el método `setResult(resultado, intent)`. Por último llamaremos al método `finish()` para finalizar la actividad. Con estas indicaciones, veamos cómo quedaría el código del botón *Aceptar*:

```
//Implementación del botón "Aceptar"
btnAceptar.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View arg0) {
            //Guardamos el mensaje personalizado en las preferencias
            SharedPreferences prefs =
                getSharedPreferences("WidgetPrefs", Context.MODE_PRIVATE);
            SharedPreferences.Editor editor = prefs.edit();

            editor.putString("msg_" + widgetId,
```

```
                                txtMensaje.getText().toString());
             editor.commit();

             //Actualizamos el widget tras la configuración
             AppWidgetManager appWidgetManager =
                 AppWidgetManager.getInstance(WidgetConfig.this);
             MiWidget.actualizarWidget(WidgetConfig.this,
                                       appWidgetManager, widgetId);

             //Devolvemos como resultado: ACEPTAR (RESULT_OK)
             Intent resultado = new Intent();
             resultado.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, widgetId);
             setResult(RESULT_OK, resultado);
             finish();
         }
});
```

Ya hemos terminado de implementar nuestra actividad de configuración. Pero para su correcto funcionamiento aún nos quedan dos detalles más por modificar. En primer lugar tendremos que declarar esta actividad en nuestro fichero `AndroidManifest.xml`, indicando que debe responder a los mensajes de tipo `APPWIDGET_CONFIGURE`:

```
<activity android:name=".WidgetConfig">
    <intent-filter>
        <action android:name="android.apwidget.action.APPWIDGET_CONFIGURE"/>
    </intent-filter>
</activity>
```

Por último, debemos indicar en el XML de configuración de nuestro widget (`xml\miwidget_wprovider.xml`) que al añadir una instancia de este widget debe mostrarse la actividad de configuración que hemos creado. Esto se consigue estableciendo el atributo `android:configure` del provider. Aprovecharemos además este paso para establecer el tiempo de actualización automática del widget al mínimo permitido por este parámetro (30 minutos) y el tamaño del widget a 2x2 celdas. Veamos cómo quedaría finalmente:

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:initialLayout="@layout/miwidget"
    android:minWidth="146dip"
    android:minHeight="146dip"
    android:label="Mi Primer Widget"
    android:updatePeriodMillis="1800000"
    android:configure="net.sgoliver.android.WidgetConfig"
/>
```

Con esto, ya tenemos todo listo para que al añadir nuestro widget al escritorio se muestre automáticamente la pantalla de configuración que hemos construido. Podemos ejecutar el proyecto en este punto y comprobar que todo funciona correctamente.

Como siguiente paso vamos a modificar el layout del widget que ya construimos en el apartado anterior para añadir una nueva etiqueta de texto donde mostraremos la hora actual, y un botón que nos servirá para forzar la actualización de los datos del widget:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="5dip">

    <FrameLayout
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:background="#000000"
        android:padding="10dip" >

        <LinearLayout
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:background="#FFFFFF"
            android:padding="5dip"
            android:orientation="vertical">

            <TextView android:id="@+id/LblMensaje"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:textColor="#000000"
                android:text="mensaje" />

            <TextView android:id="@+id/LblHora"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:textColor="#000000"
                android:text="hora_actual" />

            <Button android:id="@+id/BtnActualizar"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="Actualizar" />

        </LinearLayout>

    </FrameLayout>

</LinearLayout>
```

Hecho esto, tendremos que modificar la implementación de nuestro provider (`MiWidget.java`) para que en cada actualización del widget se actualicen sus controles con los datos correctos (recordemos que en el apartado anterior dejamos este evento de actualización vacío ya que no mostrábamos datos actualizables en el widget). Esto lo haremos dentro del evento `onUpdate()` de nuestro provider.

Como ya dijimos, los componentes de un widget se basan en un tipo especial de vistas que llamamos *Remote Views*. Pues bien, para acceder a la lista de estos componentes que constituyen la interfaz del widget construiremos un nuevo objeto `RemoteViews` a partir del ID del layout de cada widget. Obtenida la lista de componentes, tendremos disponibles una serie de métodos set (uno para cada tipo de datos básicos) para establecer las propiedades de cada control del widget. Estos métodos reciben como parámetros el ID del control, el nombre del método que queremos ejecutar sobre el control, y el valor a establecer. Además de estos métodos, contamos adicionalmente con una serie de métodos más específicos para establecer directamente el texto y otras propiedades sencillas de los controles `TextView`, `ImageView`, `ProgressBar` y `Chronometer`, como por ejemplo `setTextViewText(idControl, valor)` para establecer el texto de un control

`TextView`. Pueden consultarse todos los métodos disponibles en la [documentación oficial](#) de la clase `RemoteViews`. De esta forma, si por ejemplo queremos establecer el texto del control cuyo id es `LblMensaje` haríamos lo siguiente:

```
RemoteViews controles = new RemoteViews(context.getPackageName(),
R.layout.miwidget);
controles.setTextViewText(R.id.LblMensaje, "Mensaje de prueba");
```

El proceso de actualización habrá que realizarlo por supuesto para todas las instancias del widget que se hayan añadido al escritorio. Recordemos aquí que el evento `onUpdate()` recibe como parámetro la lista de widgets que hay que actualizar.

Dicho esto, creo que ya podemos mostrar cómo quedaría el código de actualización de nuestro widget:

```
@Override
public void onUpdate(Context context,
                    AppWidgetManager appWidgetManager,
                    int[] appWidgetIds) {

    //Iteramos la lista de widgets en ejecución
    for (int i = 0; i < appWidgetIds.length; i++)
    {
        //ID del widget actual
        int widgetId = appWidgetIds[i];

        //Actualizamos el widget actual
        actualizarWidget(context, appWidgetManager, widgetId);
    }
}

public static void actualizarWidget(Context context,
                    AppWidgetManager appWidgetManager, int widgetId)
{
    //Recuperamos el mensaje personalizado para el widget actual
    SharedPreferences prefs =
            context.getSharedPreferences("WidgetPrefs", Context.MODE_PRIVATE);
    String mensaje = prefs.getString("msg_" + widgetId, "Hora actual:");

    //Obtenemos la lista de controles del widget actual
    RemoteViews controles =
            new RemoteViews(context.getPackageName(), R.layout.miwidget);

    //Actualizamos el mensaje en el control del widget
    controles.setTextViewText(R.id.LblMsg, mensaje);

    //Obtenemos la hora actual
    Calendar calendario = new GregorianCalendar();
    String hora = calendario.getTime().toLocaleString();

    //Actualizamos la hora en el control del widget
    controles.setTextViewText(R.id.LblHora, hora);

    //Notificamos al manager de la actualización del widget actual
    appWidgetManager.updateAppWidget(widgetId, controles);
}
```

Como vemos, todo el trabajo de actualización para un widget lo hemos extraído a un método estático independiente, de forma que también podamos llamarlo desde otras partes de la aplicación (como hacemos por ejemplo desde la actividad de configuración para forzar la primera actualización del widget).

Además quiero destacar la última línea del código, donde llamamos al método `updateAppWidget()` del *widget manager*. Esto es importante y necesario, ya que de no hacerlo la actualización de los controles no se reflejará correctamente en la interfaz del widget.

Tras esto, ya sólo nos queda implementar la funcionalidad del nuevo botón que hemos incluido en el widget para poder forzar la actualización del mismo. A los controles utilizados en los widgets de Android, que ya sabemos que son del tipo `RemoteView`, no podemos asociar eventos de la forma tradicional que hemos visto en múltiples ocasiones durante el curso. Sin embargo, en su lugar, tenemos la posibilidad de asociar a un evento (por ejemplo, el click sobre un botón) un determinado mensaje (*Pending Intent*) de tipo *broadcast* que será lanzado cada vez que se produzca dicho evento. Además, podremos configurar el widget (que como ya indicamos no es más que un componente de tipo *broadcast receiver*) para que capture esos mensajes, e implementar en el evento `onReceive()` del widget las acciones necesarias a ejecutar tras capturar el mensaje. Con estas tres acciones simularemos la captura de eventos sobre controles de un widget.

Vamos por partes. En primer lugar hagamos que se lance un intent broadcast cada vez que se pulse el botón del widget. Para ello, en el método `actualizarWidget()` construiremos un nuevo Intent asociándole una acción personalizada, que en nuestro caso llamaremos por ejemplo "`net.sgoliver.ACTUALIZAR_WIDGET`". Como parámetro del nuevo Intent insertaremos mediante putExtra() el ID del widget actual de forma que más tarde podamos saber el widget concreto que ha lanzado el mensaje. Por último crearemos el `PendingIntent` mediante el método `getBroadcast()` y lo asociaremos al evento `onClick` del control llamando a `setOnClickPendingIntent()` pasándole el ID del control, en nuestro caso el botón de "*Actualizar*". Veamos cómo queda todo esto:

```
//Asociamos los 'eventos' al widget
Intent intent = new Intent("net.sgoliver.ACTUALIZAR_WIDGET");
intent.putExtra(
    AppWidgetManager.EXTRA_APPWIDGET_ID, widgetId);

PendingIntent pendingIntent =
    PendingIntent.getBroadcast(context, widgetId,
        intent, PendingIntent.FLAG_UPDATE_CURRENT);

controles.setOnClickPendingIntent(R.id.BtnActualizar, pendingIntent);
```

Ahora vamos a declarar en el *Android Manifest* este mensaje personalizado, de forma que el widget sea capaz de capturarlo. Para ello, añadiremos simplemente un nuevo elemento `<intent-filter>` con nuestro nombre de acción personalizado:

```
<intent-filter>
    <action android:name="net.sgoliver.ACTUALIZAR_WIDGET"/>
</intent-filter>
```

Por último, vamos a implementar el evento `onReceive()` del widget para actuar en caso de recibir nuestro mensaje de actualización personalizado. Dentro de este evento comprobaremos si la acción del mensaje recibido es la nuestra, y en ese caso recuperaremos el ID del widget que lo ha lanzado, obtendremos una referencia al widget manager, y por último llamaremos nuestro método estático de actualización pasándole estos datos.

```
@Override
public void onReceive(Context context, Intent intent) {
    if (intent.getAction().equals("net.sgoliver.ACTUALIZAR_WIDGET")) {
        //Obtenemos el ID del widget a actualizar
        int widgetId = intent.getIntExtra(
            AppWidgetManager.EXTRA_APPWIDGET_ID,
            AppWidgetManager.INVALID_APPWIDGET_ID);

        //Obtenemos el widget manager de nuestro contexto
        AppWidgetManager widgetManager =
            AppWidgetManager.getInstance(context);

        //Actualizamos el widget
        if (widgetId != AppWidgetManager.INVALID_APPWIDGET_ID) {
            actualizarWidget(context, widgetManager, widgetId);
        }
}
```

Con esto, por fin, hemos ya finalizado la construcción de nuestro widget Android y podemos ejecutar el proyecto de Eclipse para comprobar que todo funciona correctamente, tanto para una sola instancia como para varias instancias simultaneas.

Un comentario final, la actualización automática del widget se ha establecido a la frecuencia mínima que permite el atributo `updatePeriodMillis` del widget provider, que son 30 minutos. Por tanto es difícil y aburrido esperar para verla en funcionamiento mientras probamos el widget en el emulador. Pero funciona, os lo aseguro. De cualquier forma, esos 30 minutos pueden ser un periodo demasiado largo de tiempo según la funcionalidad que queramos dar a nuestro widget, que puede requerir tiempos de actualización mucho más cortos (ojo con el rendimiento y el gasto de batería). Más adelante, cuando hablemos de Alarmas, veremos una técnica que nos permitirá actualizar los widgets sin esa limitación de 30 minutos. Hasta entonces, espero que el tema os sea de utilidad y que os haya parecido interesante.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-widgets-2.zip



Menús

IV. Menús

Menús y Submenús básicos

En los dos siguientes apartados del Curso de Programación Android nos vamos a centrar en la creación de menús de opciones en sus diferentes variantes.

En Android podemos encontrar 3 tipos diferentes de menús:

- *Menús Principales*. Los más habituales, aparecen en la zona inferior de la pantalla al pulsar el botón 'menú' del teléfono.
- *Submenús*. Son menús secundarios que se pueden mostrar al pulsar sobre una opción de un menú principal.
- *Menús Contextuales*. Útiles en muchas ocasiones, aparecen al realizar una pulsación larga sobre algún elemento de la pantalla.

En este primer apartado sobre el tema veremos cómo trabajar con los dos primeros tipos de menús. En el siguiente, comentaremos los menús contextuales y algunas características más avanzadas.

Como casi siempre, vamos a tener dos alternativas a la hora de mostrar un menú en nuestra aplicación Android. La primera de ellas mediante la definición del menú en un fichero XML, y la segunda creando el menú directamente mediante código. En este apartado veremos ambas alternativas.

Veamos en primer lugar cómo crear un menú a partir de su definición en XML. Los ficheros XML de menú se deben colocar en la carpeta "`res\menu`" de nuestro proyecto y tendrán una estructura análoga a la del siguiente ejemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<menu
  xmlns:android="http://schemas.android.com/apk/res/android">

<item android:id="@+id/MnuOpc1" android:title="Opcion1"
      android:icon="@drawable/tag"></item>
<item android:id="@+id/MnuOpc2" android:title="Opcion2"
      android:icon="@drawable/filter"></item>
<item android:id="@+id/MnuOpc3" android:title="Opcion3"
      android:icon="@drawable/chart"></item>

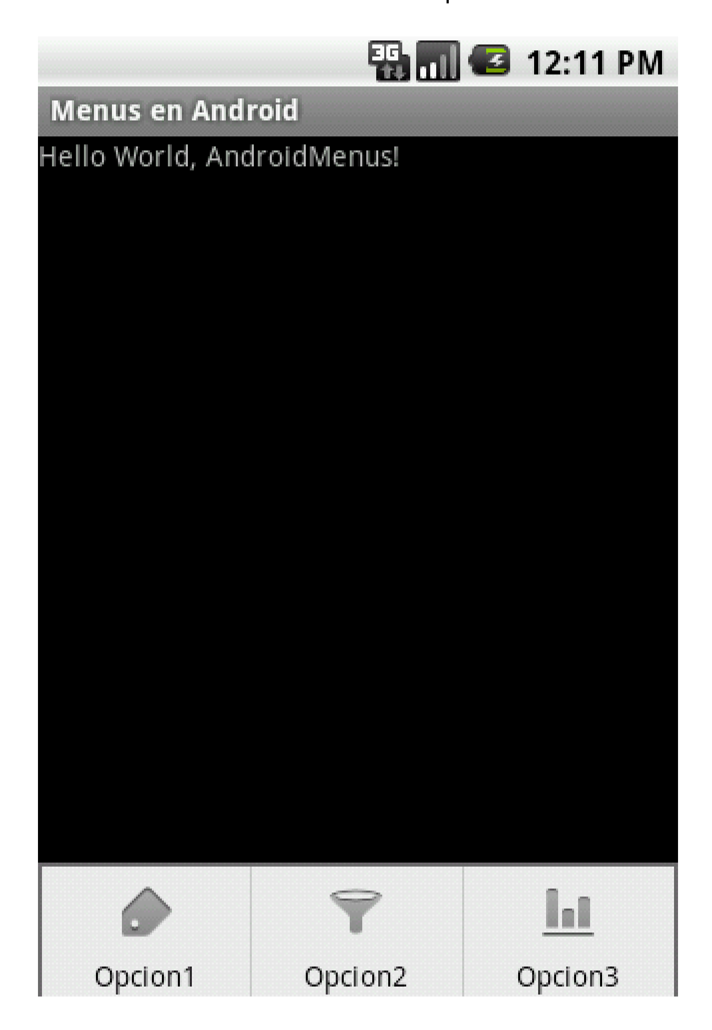
</menu>
```

Como vemos, la estructura básica de estos ficheros es muy sencilla. Tendremos un elemento principal `<menu>` que contendrá una serie de elementos `<item>` que se corresponderán con las distintas opciones a mostrar en el menú. Estos elementos `<item>` tendrán a su vez varias propiedades básicas, como su ID (`android:id`), su texto (`android:title`) o su icono (`android:icon`). Los iconos utilizados deberán estar por supuesto en las carpetas "`res\drawable-`…" de nuestro proyecto (al final del apartado os paso unos enlaces donde podéis conseguir algunos iconos de menú Android gratuitos).

Una vez definido el menú en el fichero XML, tendremos que implementar el evento onCreateOptionsMenu() de la actividad que queremos que lo muestre. En este evento deberemos "inflar" el menú de forma parecida a como ya hemos hecho otras veces con otro tipo de layouts. Primero obtendremos una referencia al *inflater* mediante el método getMenuInflater() y posteriormente generaremos la estructura del menú llamando a su método infate() pasándole como parámetro el ID del menú definido en XML, que en nuestro caso será R.menu.menu_principal. Por último devolveremos el valor true para confirmar que debe mostrarse el menú.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    //Alternativa 1
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_principal, menu);
    return true;
}
```

Y ya hemos terminado, con estos sencillos pasos nuestra aplicación ya debería mostrar sin problemas el menú que hemos construido, aunque todavía nos faltaría implementar la funcionalidad de cada una de las opciones mostradas.



Como hemos comentado antes, este mismo menú también lo podríamos crear directamente mediante código, también desde el evento `onCreateOptionsMenu()`. Para ello, para añadir cada opción del menú podemos utilizar el método `add()` sobre el objeto de tipo `Menu` que nos llega como parámetro del evento. Este método recibe 4 parámetros: ID del grupo asociado a la opción (veremos qué es esto en el siguiente apartado, por ahora utilizaremos `Menu.NONE`), un ID único para la opción (que declararemos como constantes de la clase), el orden de la opción (que no nos interesa por ahora, utilizaremos `Menu.NONE`) y el texto de la opción. Por otra parte, el icono de cada opción lo estableceremos mediante el método `setIcon()` pasándole el ID del recurso.

Veamos cómo quedaría el código utilizando esta alternativa, que generaría un menú exactamente igual al del ejemplo anterior:

```
private static final int MNU_OPC1 = 1;
private static final int MNU_OPC2 = 2;
private static final int MNU_OPC3 = 3;

//...

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    //Alternativa 2
    menu.add(Menu.NONE, MNU_OPC1, Menu.NONE,
"Opcion1").setIcon(R.drawable.tag);
    menu.add(Menu.NONE, MNU_OPC1, Menu.NONE,
"Opcion2").setIcon(R.drawable.filter);
    menu.add(Menu.NONE, MNU_OPC1, Menu.NONE,
"Opcion3").setIcon(R.drawable.chart);
    return true;
}
```

Construido el menú, la implementación de cada una de las opciones se incluirá en el evento `onOptionsItemSelected()` de la actividad que mostrará el menú. Este evento recibe como parámetro el item de menú que ha sido pulsado por el usuario, cuyo ID podemos recuperar con el método `getItemId()`. Según este ID podremos saber qué opción ha sido pulsada y ejecutar unas acciones u otras. En nuestro caso de ejemplo, lo único que haremos será modificar el texto de una etiqueta (`lblMensaje`) colocada en la pantalla principal de la aplicación.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.MnuOpc1:
            lblMensaje.setText("Opcion 1 pulsada!");
            return true;
        case R.id.MnuOpc2:
            lblMensaje.setText("Opcion 2 pulsada!");;
            return true;
        case R.id.MnuOpc3:
            lblMensaje.setText("Opcion 3 pulsada!");;
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Con esto, hemos conseguido ya un menú completamente funcional. Si ejecutamos el proyecto en el emulador comprobaremos cómo al pulsar el botón de 'menú' del teléfono aparece el menú que hemos definido y que al pulsar cada opción se muestra el mensaje de ejemplo.

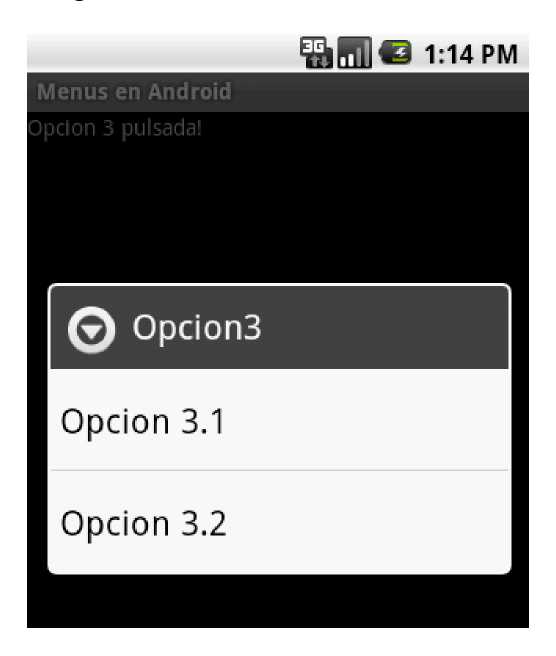
Pasemos ahora a comentar los submenús. Un submenú no es más que un menú secundario que se muestra al pulsar una opción determinada de un menú principal. Los submenús en Android se muestran en forma de lista emergente, cuyo título contiene el texto de la opción elegida en el menú principal. Como ejemplo, vamos a añadir un submenú a la Opción 3 del ejemplo anterior, al que añadiremos dos nuevas opciones secundarias. Para ello, bastará con insertar en el XML de menú un nuevo elemento `<menu>` dentro del item correspondiente a la opción 3. De esta forma, el XML quedaría ahora como sigue:

```
<?xml version="1.0" encoding="utf-8"?>
<menu
  xmlns:android="http://schemas.android.com/apk/res/android">

<item android:id="@+id/MnuOpc1" android:title="Opcion1"
      android:icon="@drawable/tag"></item>
<item android:id="@+id/MnuOpc2" android:title="Opcion2"
      android:icon="@drawable/filter"></item>
<item android:id="@+id/MnuOpc3" android:title="Opcion3"
      android:icon="@drawable/chart">
    <menu>
        <item android:id="@+id/SubMnuOpc1"
              android:title="Opcion 3.1" />
        <item android:id="@+id/SubMnuOpc2"
              android:title="Opcion 3.2" />
    </menu>
</item>

</menu>
```

Si volvemos a ejecutar ahora el proyecto y pulsamos la opción 3 nos aparecerá el correspondiente submenú con las dos nuevas opciones añadidas. Lo vemos en la siguiente imagen:



Comprobamos como efectivamente aparecen las dos nuevas opciones en la lista emergente, y que el título de la lista se corresponde con el texto de la opción elegida en el menú principal ("Opcion3").

Para conseguir esto mismo mediante código procederíamos de forma similar a la anterior, con la única diferencia de que la opción de menú 3 la añadiremos utilizando el método `addSubMenu()` en vez de `add()`, y guardando una referencia al submenú. Sobre el submenú añadido insertaremos las dos nuevas opciones utilizando una vez más el método `add()`. Vemos cómo quedaría:

```
//Alternativa 2
menu.add(Menu.NONE, MNU_OPC1, Menu.NONE, "Opcion1").setIcon(R.drawable.tag);
menu.add(Menu.NONE, MNU_OPC1, Menu.NONE,
"Opcion2").setIcon(R.drawable.filter);
//menu.add(Menu.NONE, MNU_OPC1, Menu.NONE,
"Opcion3").setIcon(R.drawable.chart);

SubMenu smnu = menu.addSubMenu(Menu.NONE, MNU_OPC1, Menu.NONE, "Opcion3")
                            .setIcon(R.drawable.chart);
smnu.add(Menu.NONE, SMNU_OPC1, Menu.NONE, "Opcion 3.1");
smnu.add(Menu.NONE, SMNU_OPC2, Menu.NONE, "Opcion 3.2");
```

En cuanto a la implementación de estas opciones de submenú no habría diferencia con todo lo comentado anteriormente ya que también se tratan desde el evento `onOptionsItemSelected()`, identificándolas por su ID.

Por tanto, con esto habríamos terminado de comentar las opciones básicas a la hora de crear menús y submenús en nuestras aplicaciones Android. En el siguiente apartado veremos algunas opciones algo más avanzadas que, aunque menos frecuentes, puede que nos hagan falta para desarrollar determinadas aplicaciones.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-menus-1.zip

Si necesitáis iconos para mostrar en los menús aquí tenéis varios enlaces con algunos gratuitos que podéis utilizar en vuestras aplicaciones Android:

- http://www.androidicons.com/freebies.php
- http://www.glyfx.com/products/free_android2.html

Menús Contextuales

En el apartado anterior del curso ya vimos cómo crear menús y submenús básicos para nuestras aplicaciones Android. Sin embargo, existe otro tipo de menús que nos pueden ser muy útiles en determinados contextos: los *menús contextuales*. Este tipo de menú siempre va asociado a un control concreto de la pantalla y se muestra al realizar una pulsación larga sobre éste. Suele mostrar opciones específicas disponibles únicamente para el elemento pulsado. Por ejemplo, en un control de tipo lista podríamos tener un menú contextual que apareciera al pulsar sobre un elemento concreto de la lista y que permitiera editar su texto o eliminarlo de la colección.

Pues bien, la creación y utilización de este tipo de menús es muy parecida a lo que ya vimos para los menús y submenús básicos, pero presentan algunas particularidades que hacen interesante tratarlos al margen del resto en este nuevo apartado.

Empecemos por un caso sencillo. Vamos a partir del ejemplo del apartado anterior, al que vamos a añadir en primer lugar un menú contextual que aparezca al pulsar sobre la etiqueta de texto que mostrábamos en la ventana principal de la aplicación. Para ello, lo primero que vamos a hacer es indicar en el método `onCreate()` de nuestra actividad principal que la etiqueta tendrá asociado un menú contextual. Esto lo conseguimos con una llamada a `registerForContextMenu()`:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    //Obtenemos las referencias a los controles
    lblMensaje = (TextView)findViewById(R.id.LblMensaje);

    //Asociamos los menús contextuales a los controles
    registerForContextMenu(lblMensaje);
}
```

A continuación, igual que hacíamos con `onCreateOptionsMenu()` para los menús básicos, vamos a sobrescribir en nuestra actividad el evento encargado de construir los menús contextuales asociados a los diferentes controles de la aplicación. En este caso el evento se llama `onCreateContextMenu()`, y a diferencia de `onCreateOptionsMenu()` éste se llama cada vez que se necesita mostrar un menú contextual, y no una sola vez al inicio de la aplicación. En este evento actuaremos igual que para los menús básicos, inflando el menú XML que hayamos creado con las distintas opciones, o creando a mano el menú mediante el método `add()` [para más información leer el apartado anterior].

En nuestro ejemplo hemos definido un menú en XML llamado "`menu_ctx_etiqueta.xml`":

```
<?xml version="1.0" encoding="utf-8"?>
<menu
  xmlns:android="http://schemas.android.com/apk/res/android">

<item android:id="@+id/CtxLblOpc1"
    android:title="OpcEtiqueta1"></item>
<item android:id="@+id/CtxLblOpc2"
    android:title="OpcEtiqueta2"></item>

</menu>
```

Por su parte el evento `onCreateContextMenu()` quedaría de la siguiente forma:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                                ContextMenuInfo menuInfo)
{
    super.onCreateContextMenu(menu, v, menuInfo);

    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_ctx_etiqueta, menu);
}
```

Por último, para implementar las acciones a realizar tras pulsar una opción determinada del menú contextual vamos a implementar el evento `onContextItemSelected()` de forma análoga a cómo hacíamos con `onOptionsItemSelected()` para los menús básicos:

```
@Override
public boolean onContextItemSelected(MenuItem item) {

    switch (item.getItemId()) {
        case R.id.CtxLblOpc1:
            lblMensaje.setText("Etiqueta: Opcion 1 pulsada!");
            return true;
        case R.id.CtxLblOpc2:
            lblMensaje.setText("Etiqueta: Opcion 2 pulsada!");
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

Con esto, ya tendríamos listo nuestro menú contextual para la etiqueta de texto de la actividad principal, y como veis todo es prácticamente análogo a cómo construimos los menús y submenús básicos en el apartado anterior. En este punto ya podríamos ejecutar el proyecto en el emulador y comprobar su funcionamiento.

Ahora vamos con algunas particularidades. Los menús contextuales se utilizan a menudo con controles de tipo lista, lo que añade algunos detalles que conviene mencionar. Para ello vamos a añadir a nuestro ejemplo una lista con varios datos de muestra y vamos a asociarle un nuevo menú contextual. Modificaremos el layout XML de la ventana principal para añadir el control `ListView` y modificaremos el método `onCreate()` para obtener la referencia al control, insertar varios datos de ejemplo, y asociarle un menú contextual:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    //Obtenemos las referencias a los controles
    lblMensaje = (TextView)findViewById(R.id.LblMensaje);
    lstLista = (ListView)findViewById(R.id.LstLista);

    //Rellenamos la lista con datos de ejemplo
    String[] datos =
        new String[]{"Elem1","Elem2","Elem3","Elem4","Elem5"};


    ArrayAdapter<String> adaptador =
        new ArrayAdapter<String>(this,
```

```
            android.R.layout.simple_list_item_1, datos);

    lstLista.setAdapter(adaptador);

    //Asociamos los menús contextuales a los controles
    registerForContextMenu(lblMensaje);
    registerForContextMenu(lstLista);
}
```

Como en el caso anterior, vamos a definir en XML otro menú para asociarlo a los elementos de la lista, lo llamaremos "`menu_ctx_lista`":

```
<?xml version="1.0" encoding="utf-8"?>
<menu
  xmlns:android="http://schemas.android.com/apk/res/android">

<item android:id="@+id/CtxLstOpc1"
    android:title="OpcLista1"></item>
<item android:id="@+id/CtxLstOpc2"
    android:title="OpcLista2"></item>

</menu>
```

Como siguiente paso, y dado que vamos a tener varios menús contextuales en la misma actividad, necesitaremos modificar el evento `onCreateContextMenu()` para que se construya un menú distinto dependiendo del control asociado. Esto lo haremos obteniendo el ID del control al que se va a asociar el menú contextual, que se recibe en forma de parámetro (`View v`) en el evento `onCreateContextMenu()`.

Utilizaremos para ello una llamada al método `getId()` de dicho parámetro:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                                ContextMenuInfo menuInfo)
{
    super.onCreateContextMenu(menu, v, menuInfo);

    MenuInflater inflater = getMenuInflater();

    if(v.getId() == R.id.LblMensaje)
        inflater.inflate(R.menu.menu_ctx_etiqueta, menu);
    else if(v.getId() == R.id.LstLista)
    {
        AdapterView.AdapterContextMenuInfo info =
            (AdapterView.AdapterContextMenuInfo)menuInfo;

        menu.setHeaderTitle(
            lstLista.getAdapter().getItem(info.position).toString());

        inflater.inflate(R.menu.menu_ctx_lista, menu);
    }
}
```

Vemos cómo en el caso del menú para el control lista hemos ido además un poco más allá, y hemos personalizado el título del menú contextual [mediante `setHeaderTitle()`] para que muestre el texto del elemento seleccionado en la lista. Para hacer esto nos hace falta

saber la posición en la lista del elemento seleccionado, algo que podemos conseguir haciendo uso del último parámetro recibido en el evento onCreateContextMenu(), llamado menuInfo. Este parámetro contiene información adicional del control que se ha pulsado para mostrar el menú contextual, y en el caso particular del control ListView contiene la posición del elemento concreto de la lista que se ha pulsado. Para obtenerlo, convertimos el parámetro menuInfo a un objeto de tipo AdapterContextMenuInfo y accedemos a su atributo position tal como vemos en el código anterior.

La respuesta a este nuevo menú se realizará desde el mismo evento que el anterior, todo dentro de onContextItemSelected(). Por tanto, incluyendo las opciones del nuevo menú contextual para la lista el código nos quedaría de la siguiente forma:

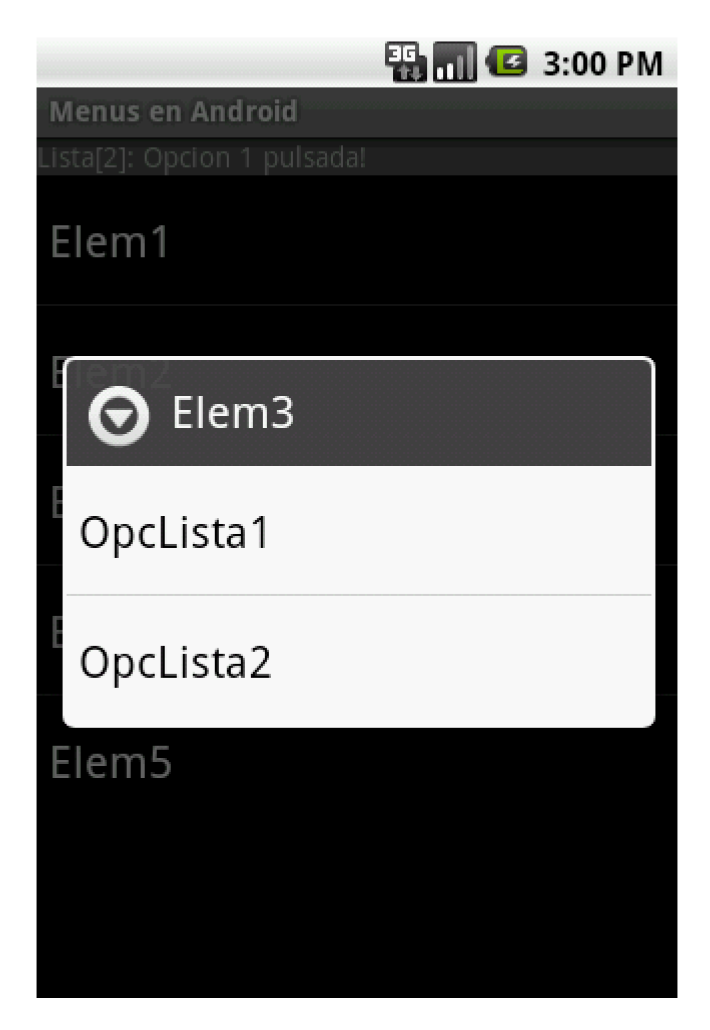
```
@Override
public boolean onContextItemSelected(MenuItem item) {

    AdapterContextMenuInfo info =
        (AdapterContextMenuInfo) item.getMenuInfo();

    switch (item.getItemId()) {
        case R.id.CtxLblOpc1:
            lblMensaje.setText("Etiqueta: Opcion 1 pulsada!");
            return true;
        case R.id.CtxLblOpc2:
            lblMensaje.setText("Etiqueta: Opcion 2 pulsada!");
            return true;
        case R.id.CtxLstOpc1:
            lblMensaje.setText(
                        "Lista[" + info.position + "]: Opcion 1 pulsada!");
            return true;
        case R.id.CtxLstOpc2:
            lblMensaje.setText(
                        "Lista[" + info.position + "]: Opcion 2 pulsada!");
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

Como vemos, aquí también utilizamos la información del objeto AdapterContextMenuInfo para saber qué elemento de la lista se ha pulsado, con la única diferencia de que en esta ocasión lo obtenemos mediante una llamada al método getMenuInfo() de la opción de menú (MenuItem) recibida como parámetro.

Si volvemos a ejecutar el proyecto en este punto podremos comprobar el aspecto de nuestro menú contextual al pulsar cualquier elemento de la lista:



A modo de resumen, en este apartado hemos visto cómo crear menús contextuales asociados a determinados elementos y controles de nuestra interfaz de la aplicación. Hemos visto cómo crear menús básicos y algunas particularidades que existen a la hora de asociar menús contextuales a elementos de un control de tipo lista. Para no alargar este apartado dedicaremos un tercero a comentar algunas opciones menos frecuentes, pero igualmente útiles, de los menús en Android.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-menus-2.zip

Opciones avanzadas de menú

En los apartados anteriores del curso ya hemos visto cómo crear menús básicos para nuestras aplicaciones, tanto menús principales como de tipo contextual. Sin embargo, se nos quedaron en el tintero un par de temas que también nos pueden ser necesarios o interesantes a la hora de desarrollar una aplicación. Por un lado veremos los *grupos de opciones*, y por otro la *actualización dinámica* de un menú según determinadas condiciones.

Los grupos de opciones son un mecanismo que nos permite agrupar varios elementos de un menú de forma que podamos aplicarles ciertas acciones o asignarles determinadas características o funcionalidades de forma conjunta. De esta forma, podremos por ejemplo

habilitar o deshabilitar al mismo tiempo un grupo de opciones de menú, o hacer que sólo se pueda seleccionar una de ellas. Lo veremos más adelante.

Veamos primero cómo definir un grupo de opciones de menú. Como ya comentamos, Android nos permite definir un menú de dos formas distintas: mediante un fichero XML, o directamente a través de código. Si elegimos la primera opción, para definir un grupo de opciones nos basta con colocar dicho grupo dentro de un elemento <group>, al que asignaremos un ID. Veamos un ejemplo. Vamos a definir un menú con 3 opciones principales, donde la última opción abre un submenú con 2 opciones que formen parte de un grupo. A todas las opciones le asignaremos un ID y un texto, y a las opciones principales asignaremos además una imagen.

```
<menu
  xmlns:android="http://schemas.android.com/apk/res/android">

<item android:id="@+id/MnuOpc1" android:title="Opcion1"
      android:icon="@drawable/tag"></item>
<item android:id="@+id/MnuOpc2" android:title="Opcion2"
      android:icon="@drawable/filter"></item>
<item android:id="@+id/MnuOpc3" android:title="Opcion3"
      android:icon="@drawable/chart">
    <menu>
        <group android:id="@+id/grupo1">

            <item android:id="@+id/SubMnuOpc1"
                  android:title="Opcion 3.1" />
            <item android:id="@+id/SubMnuOpc2"
                  android:title="Opcion 3.2" />

        </group>
    </menu>
</item>

</menu>
```

Como vemos, las dos opciones del submenú se han incluido dentro de un elemento <group>. Esto nos permitirá ejecutar algunas acciones sobre todas las opciones del grupo de forma conjunta, por ejemplo deshabilitarlas u ocultarlas:

```
//Deshabilitar todo el grupo
mnu.setGroupEnabled(R.id.grupo1, false);

//Ocultar todo el grupo
mnu.setGroupVisible(R.id.grupo1, false);
```

Además de estas acciones, también podemos modificar el comportamiento de las opciones del grupo de forma que tan sólo se pueda seleccionar una de ellas, o para que se puedan seleccionar varias. Con esto convertiríamos el grupo de opciones de menú en el equivalente a un conjunto de controles RadioButton o CheckBox respectivamente. Esto lo conseguimos utilizando el atributo android:checkableBehavior del elemento <group>, al que podemos asignar el valor "single" (selección exclusiva, tipo RadioButton) o "all" (selección múltiple, tipo CheckBox). En nuestro caso de ejemplo vamos a hacer seleccionable sólo una de las opciones del grupo:

```
<group android:id="@+id/grupo1" android:checkableBehavior="single">

    <item android:id="@+id/SubMnuOpc1"
        android:title="Opcion 3.1" />
    <item android:id="@+id/SubMnuOpc2"
        android:title="Opcion 3.2" />

</group>
```

Si optamos por construir el menú directamente mediante código debemos utilizar el método `setGroupCheckable()` al que pasaremos como parámetros el ID del grupo y el tipo de selección que deseamos (exclusiva o no). Así, veamos el método de construcción del menú anterior mediante código:

```
private static final int MNU_OPC1 = 1;
private static final int MNU_OPC2 = 2;
private static final int MNU_OPC3 = 3;
private static final int SMNU_OPC1 = 31;
private static final int SMNU_OPC2 = 32;

private static final int GRUPO_MENU_1 = 101;

private int opcionSeleccionada = 0;

//...

private void construirMenu(Menu menu)
{
     menu.add(Menu.NONE, MNU_OPC1, Menu.NONE,
               "Opcion1").setIcon(R.drawable.tag);
     menu.add(Menu.NONE, MNU_OPC2, Menu.NONE,
               "Opcion2").setIcon(R.drawable.filter);

     SubMenu smnu = menu.addSubMenu(Menu.NONE, MNU_OPC3, Menu.NONE, "Opcion3")
                              .setIcon(R.drawable.chart);

     smnu.add(GRUPO_MENU_1, SMNU_OPC1, Menu.NONE, "Opcion 3.1");
     smnu.add(GRUPO_MENU_1, SMNU_OPC2, Menu.NONE, "Opcion 3.2");

     //Establecemos la selección exclusiva para el grupo de opciones
     smnu.setGroupCheckable(GRUPO_MENU_1, true, true);

     //Marcamos la opción seleccionada actualmente
     if(opcionSeleccionada == 1)
          smnu.getItem(0).setChecked(true);
     else if(opcionSeleccionada == 2)
          smnu.getItem(1).setChecked(true);
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {

     construirMenu(menu);

     return true;
}
```

Como vemos, al final del método nos ocupamos de marcar manualmente la opción seleccionada actualmente, que debemos conservar en algún atributo interno (en mi caso lo he llamado `opcionSeleccionada`) de nuestra actividad. Esta marcación manual la hacemos mediante el método `getItem()` para obtener una opción determinada del submenú y sobre

ésta el método `setChecked()` para establecer su estado. ¿Por qué debemos hacer esto? ¿No guarda Android el estado de las opciones de menú seleccionables? La respuesta es sí, sí lo hace, pero siempre que no reconstruyamos el menú entre una visualización y otra. ¿Pero no dijimos que la creación del menú sólo se realiza una vez en la primera llamada a `onCreateOptionsMenu()`? También es cierto, pero después veremos cómo también es posible preparar nuestra aplicación para poder modificar de forma dinámica un menú según determinadas condiciones, lo que sí podría implicar reconstruirlo previamente a cada visualización. En definitiva, si guardamos y restauramos nosotros mismos el estado de las opciones de menú seleccionables estaremos seguros de no perder su estado bajo ninguna circunstancia.

Por supuesto, para mantener el estado de las opciones hará falta actualizar el atributo `opcionSeleccionada` tras cada pulsación a una de las opciones. Esto lo haremos como siempre en el método `onOptionItemSelected()`.

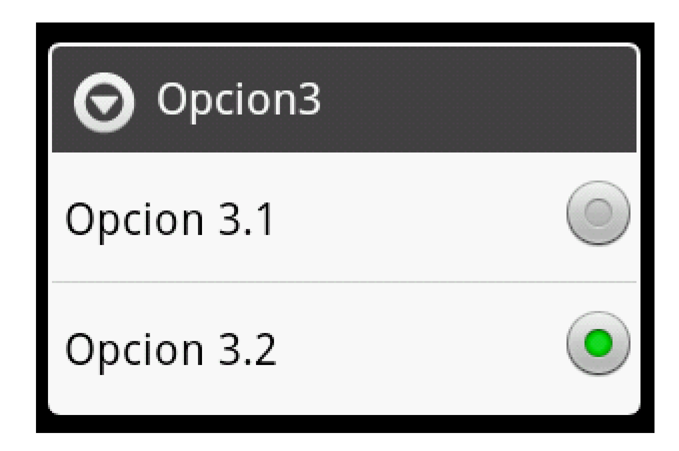
```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {

        //...
        //Omito el resto de opciones por simplicidad

        case SMNU_OPC1:
            opcionSeleccionada = 1;
            item.setChecked(true);
            return true;
        case SMNU_OPC2:
            opcionSeleccionada = 2;
            item.setChecked(true);
            return true;

        //...
    }
}
```

Con esto ya podríamos probar cómo nuestro menú funciona de la forma esperada, permitiendo marcar sólo una de las opciones del submenú. Si visualizamos y marcamos varias veces distintas opciones veremos cómo se mantiene correctamente el estado de cada una de ellas entre diferentes llamadas.



El segundo tema que quería desarrollar en este apartado trata sobre la modificación dinámica de un menú durante la ejecución de la aplicación de forma que éste sea distinto según determinadas condiciones. Supongamos por ejemplo que normalmente vamos a querer mostrar nuestro menú con 3 opciones, pero si tenemos marcada en pantalla una determinada opción queremos mostrar en el menú una opción adicional. ¿Cómo hacemos esto si dijimos que el evento `onCreateOptionsMenu()` se ejecuta una sola vez? Pues esto es posible ya que además del evento indicado existe otro llamado `onPrepareOptionsMenu()` que se ejecuta cada vez que se va a mostrar el menú de la aplicación, con lo que resulta el lugar ideal para adaptar nuestro menú a las condiciones actuales de la aplicación.

Para mostrar el funcionamiento de esto vamos a colocar en nuestra aplicación de ejemplo un nuevo *checkbox* (lo llamaré en mi caso `chkMenuExtendido`). Nuestra intención es que si este checkbox está marcado el menú muestre una cuarta opción adicional, y en caso contrario sólo muestre las tres opciones ya vistas en los ejemplos anteriores.

En primer lugar prepararemos el método `construirMenu()` para que reciba un parámetro adicional que indique si queremos construir un menú extendido o no, y sólo añadiremos la cuarta opción si este parámetro llega activado.

```
private void construirMenu(Menu menu, boolean extendido)
{
     menu.add(Menu.NONE, MNU_OPC1, Menu.NONE,
              "Opcion1").setIcon(R.drawable.tag);
     menu.add(Menu.NONE, MNU_OPC2, Menu.NONE,
              "Opcion2").setIcon(R.drawable.filter);

     SubMenu smnu = menu.addSubMenu(Menu.NONE, MNU_OPC3, Menu.NONE, "Opcion3")
                              .setIcon(R.drawable.chart);

     smnu.add(GRUPO_MENU_1, SMNU_OPC1, Menu.NONE, "Opcion 3.1");
     smnu.add(GRUPO_MENU_1, SMNU_OPC2, Menu.NONE, "Opcion 3.2");

     //Establecemos la selección exclusiva para el grupo de opciones
     smnu.setGroupCheckable(GRUPO_MENU_1, true, true);

     if(extendido)
        menu.add(Menu.NONE, MNU_OPC4, Menu.NONE,
                 "Opcion4").setIcon(R.drawable.tag);

     //Marcamos la opción seleccionada actualmente
     if(opcionSeleccionada == 1)
          smnu.getItem(0).setChecked(true);
     else if(opcionSeleccionada == 2)
          smnu.getItem(1).setChecked(true);
}
```

Además de esto, implementaremos el evento `onPrepareOptionsMenu()` para que llame a este método de una forma u otra dependiendo del estado del nuevo *checkbox*.

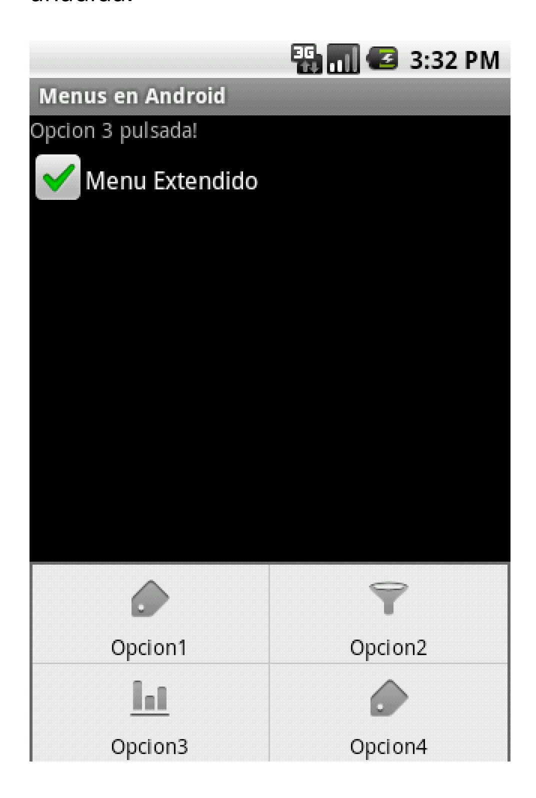
```
@Override
public boolean onPrepareOptionsMenu(Menu menu)
{
    menu.clear();

    if(chkMenuExtendido.isChecked())
        construirMenu(menu, true);
    else
        construirMenu(menu, false);

    return super.onPrepareOptionsMenu(menu);
}
```

Como vemos, en primer lugar debemos resetear el menú mediante el método `clear()` y posteriormente llamar de nuevo a nuestro método de construcción del menú indicando si queremos un menú extendido o no según el valor de la *check*.

Si ejecutamos nuevamente la aplicación de ejemplo, marcamos el checkbox y mostramos la tecla de menú podremos comprobar cómo se muestra correctamente la cuarta opción añadida.



Y con esto cerramos ya todos los temas referentes a menús que tenía intención de incluir en este Curso de Programación en Android. Espero que sea suficiente para cubrir las necesidades de muchas de vuestras aplicaciones.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-menus-avanzado.zip

5

Tratamiento de XML

V. Tratamiento de XML

Tratamiento de XML con SAX

En los siguientes apartados de este Tutorial de Desarrollo para Android vamos a comentar las distintas posibilidades que tenemos a la hora de trabajar con datos en formato XML desde la plataforma Android.

A día de hoy, en casi todas las grandes plataformas de desarrollo existen varias formas de leer y escribir datos en formato XML. Los dos modelos más extendidos son SAX (*Simple API for XML*) y DOM (*Document Object Model*). Posteriormente, han ido apareciendo otros tantos, con más o menos éxito, entre los que destaca StAX (*Streaming API for XML*). Pues bien, Android no se queda atrás en este sentido e incluye estos tres modelos principales para el tratamiento de XML, o para ser más exactos, los dos primeros como tal y una versión análoga del tercero (*XmlPull*). Por supuesto con cualquiera de los tres modelos podemos hacer las mismas tareas, pero ya veremos cómo dependiendo de la naturaleza de la tarea que queramos realizar va a resultar más eficiente utilizar un modelo u otro.

Antes de empezar, unas anotaciones respecto a los ejemplos que voy a utilizar. Estas técnicas se pueden utilizar para tratar cualquier documento XML, tanto online como local, pero por utilizar algo conocido por la mayoría de vosotros todos los ejemplos van a trabajar sobre los datos XML de un documento RSS online, concretamente sobre el canal RSS de portada de europapress.com.

Un documento RSS de este *feed* tiene la estructura siguiente:

```
<rss version="2.0">
    <channel>
        <title>Europa Press</title>
        <link>http://www.europapress.es/</link>
        <description>Noticias de Portada.</description>
        <image>
            <url>http://s01.europapress.net/eplogo.gif</url>
            <title>Europa Press</title>
            <link>http://www.europapress.es</link>
        </image>
        <language>es-ES</language>
        <copyright>Copyright</copyright>
        <pubDate>Sat, 25 Dec 2010 23:27:26 GMT</pubDate>
        <lastBuildDate>Sat, 25 Dec 2010 22:47:14 GMT</lastBuildDate>
        <item>
            <title>Título de la noticia 1</title>
            <link>http://link_de_la_noticia_2.es</link>
            <description>Descripción de la noticia 2</description>
            <guid>http://identificador_de_la_noticia_2.es</guid>
            <pubDate>Fecha de publicación 2</pubDate>
        </item>
        <item>
            <title>Título de la noticia 2</title>
            <link>http://link_de_la_noticia_2.es</link>
            <description>Descripción de la noticia 2</description>
            <guid>http://identificador_de_la_noticia_2.es</guid>
            <pubDate>Fecha de publicación 2</pubDate>
        </item>
```

```
        ...
    </channel>
</rss>
```

Como puede observarse, se compone de un elemento principal `<channel>` seguido de varios datos relativos al canal y posteriormente una lista de elementos `<item>` para cada noticia con sus datos asociados.

En estos apartados vamos a describir cómo leer este XML mediante cada una de las tres alternativas citadas, y para ello lo primero que vamos a hacer es definir una clase java para almacenar los datos de una noticia. Nuestro objetivo final será devolver una lista de objetos de este tipo, con la información de todas las noticias. Por comodidad, vamos a almacenar todos los datos como cadenas de texto:

```
public class Noticia {
    private String titulo;
    private String link;
    private String descripcion;
    private String guid;
    private String fecha;

    public String getTitulo() {
        return titulo;
    }

    public String getLink() {
        return link;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public String getGuid() {
        return guid;
    }

    public String getFecha() {
        return fecha;
    }

    public void setTitulo(String t) {
        titulo = t;
    }

    public void setLink(String l) {
        link = l;
    }

    public void setDescripcion(String d) {
        descripcion = d;
    }

    public void setGuid(String g) {
        guid = g;
    }

    public void setFecha(String f) {
        fecha = f;
    }
}
```

Una vez conocemos la estructura del XML a leer y hemos definido las clases auxiliares que nos hacen falta para almacenar los datos, pasamos ya a comentar el primero de los modelos de tratamiento de XML.

SAX en Android

En el modelo SAX, el tratamiento de un XML se basa en un analizador (*parser*) que a medida que lee secuencialmente el documento XML va generando diferentes eventos con la información de cada elemento leído. Asi, por ejemplo, a medida que lee el XML, si encuentra el comienzo de una etiqueta `<title>` generará un evento de comienzo de etiqueta, `startElement()`, con su información asociada, si después de esa etiqueta encuentra un fragmento de texto generará un evento `characters()` con toda la información necesaria, y así sucesivamente hasta el final del documento. Nuestro trabajo consistirá por tanto en implementar las acciones necesarias a ejecutar para cada uno de los eventos posibles que se pueden generar durante la lectura del documento XML.

Los principales eventos que se pueden producir son los siguientes (consultar [aquí] la lista completa):

- `startDocument()`: comienza el documento XML.
- `endDocument()`: termina el documento XML.
- `startElement()`: comienza una etiqueta XML.
- `endElement()`: termina una etiqueta XML.
- `characters()`: fragmento de texto.

Todos estos métodos están definidos en la clase `org.xml.sax.helpers.DefaultHandler`, de la cual deberemos derivar una clase propia donde se sobrescriban los eventos necesarios. En nuestro caso vamos a llamarla `RssHandler`.

```
public class RssHandler extends DefaultHandler {
    private List<Noticia> noticias;
    private Noticia noticiaActual;
    private StringBuilder sbTexto;

    public List<Noticia> getNoticias(){
        return noticias;
    }

    @Override
    public void characters(char[] ch, int start, int length)
                   throws SAXException {

        super.characters(ch, start, length);

        if (this.notciaActual != null)
            builder.append(ch, start, length);
    }

    @Override
    public void endElement(String uri, String localName, String name)
                   throws SAXException {

        super.endElement(uri, localName, name);
```

```
        if (this.notciaActual != null) {

            if (localName.equals("title")) {
                noticiaActual.setTitulo(sbTexto.toString());
            } else if (localName.equals("link")) {
                noticiaActual.setLink(sbTexto.toString());
            } else if (localName.equals("description")) {
                noticiaActual.setDescripcion(sbTexto.toString());
            } else if (localName.equals("guid")) {
                noticiaActual.setGuid(sbTexto.toString());
            } else if (localName.equals("pubDate")) {
                noticiaActual.setFecha(sbTexto.toString());
            } else if (localName.equals("item")) {
                noticias.add(noticiaActual);
            }

            sbTexto.setLength(0);
        }
    }

    @Override
    public void startDocument() throws SAXException {

        super.startDocument();

        noticias = new ArrayList<Noticia>();
        sbTexto = new StringBuilder();
    }

    @Override
    public void startElement(String uri, String localName,
                   String name, Attributes attributes) throws SAXException {

        super.startElement(uri, localName, name, attributes);

        if (localName.equals("item")) {
            noticiaActual = new Noticia();
        }
    }
}
```

Como se puede observar en el código de anterior, lo primero que haremos será incluir como miembro de la clase la lista de noticias que pretendemos construir, `List<Noticia>` noticias, y un método `getNoticias()` que permita obtenerla tras la lectura completa del documento. Tras esto, implementamos directamente los eventos SAX necesarios.

Comencemos por `startDocument()`, este evento indica que se ha comenzado a leer el documento XML, por lo que lo aprovecharemos para inicializar la lista de noticias y las variables auxiliares.

Tras éste, el evento `startElement()` se lanza cada vez que se encuentra una nueva etiqueta de apertura. En nuestro caso, la única etiqueta que nos interesará será `<item>`, momento en el que inicializaremos un nuevo objeto auxiliar de tipo Noticia donde almacenaremos posteriormente los datos de la noticia actual.

El siguiente evento relevante es `characters()`, que se lanza cada vez que se encuentra un fragmento de texto en el interior de una etiqueta. La técnica aquí será ir acumulando en una variable auxiliar, `sbTexto`, todos los fragmentos de texto que encontremos hasta detectarse una etiqueta de cierre.

Por último, en el evento de cierre de etiqueta, endElement(), lo que haremos será almacenar en el atributo apropiado del objeto noticiaActual (que conoceremos por el parámetro localName devuelto por el evento) el texto que hemos ido acumulando en la variable sbTexto y limpiaremos el contenido de dicha variable para comenzar a acumular el siguiente dato. El único caso especial será cuando detectemos el cierre de la etiqueta <item>, que significará que hemos terminado de leer todos los datos de la noticia y por tanto aprovecharemos para añadir la noticia actual a la lista de noticias que estamos construyendo.

Una vez implementado nuestro *handler*, vamos a crear una nueva clase que haga uso de él para parsear mediante SAX un documento XML concreto. A esta clase la llamaremos RssParserSax. Más adelante crearemos otras clases análogas a ésta que hagan lo mismo pero utilizando los otros dos métodos de tratamiento de XML ya mencionados. Esta clase tendrá únicamente un constructor que reciba como parámetro la URL del documento a parsear, y un método público llamado parse() para ejecutar la lectura del documento, y que devolverá como resultado una lista de noticias. Veamos cómo queda esta clase:

```
import java.io.IOException;
import java.io.InputStream;
import java.util.List;

import java.net.URL;
import javax.xml.parsers.SAXParser;
import java.net.MalformedURLException;
import javax.xml.parsers.SAXParserFactory;

public class RssParserSax
{
    private URL rssUrl;

    public RssParserSax(String url)
    {
        try
        {
            this.rssUrl = new URL(url);
        }
        catch (MalformedURLException e)
        {
            throw new RuntimeException(e);
        }
    }

    public List<Noticia> parse()
    {
        SAXParserFactory factory = SAXParserFactory.newInstance();

        try
        {
            SAXParser parser = factory.newSAXParser();
            RssHandler handler = new RssHandler();
            parser.parse(this.getInputStream(), handler);
            return handler.getNoticias();
        }
        catch (Exception e)
        {
            throw new RuntimeException(e);
        }
    }

    private InputStream getInputStream()
    {
```

```
        try
        {
            return rssUrl.openConnection().getInputStream();
        }
        catch (IOException e)
        {
            throw new RuntimeException(e);
        }
    }
}
```

Como se puede observar en el código anterior, el constructor de la clase se limitará a aceptar como parámetro la URL del documento XML a parsear a controlar la validez de dicha URL, generando una excepción en caso contrario.

Por su parte, el método `parse()` será el encargado de crear un nuevo parser SAX mediante sú fábrica correspondiente [lo que se consigue obteniendo una instancia de la fábrica con `SAXParserFactory.newInstance()` y creando un nuevo parser con `factory.newSaxParser()`] y de iniciar el proceso pasando al parser una instancia del *handler* que hemos creado anteriormente y una referencia al documento a parsear en forma de *stream*.

Para esto último, nos apoyamos en un método privado auxiliar `getInputStream()`, que se encarga de abrir la conexión con la URL especificada [mediante `openConnection()`] y obtener el *stream* de entrada [mediante `getInputStream()`].

Con esto ya tenemos nuestra aplicación Android preparada para parsear un documento XML online utilizando el modelo SAX. Veamos lo simple que sería ahora llamar a este parser por ejemplo desde nuestra actividad principal:

```
public void onCreate(Bundle savedInstanceState)
{
     super.onCreate(savedInstanceState);
     setContentView(R.layout.main);

     RssParserSax saxparser =
            new RssParserSax("http://www.europapress.es/rss/rss.aspx");

     List<Noticia> noticias = saxparser.parse();

     //Manipulación del array de noticias
     //...
}
```

Las líneas 6 y 9 del código anterior son las que hacen toda la magia. Primero creamos el parser SAX pasándole la URL del documento XML y posteriormente llamamos al método `parse()` para obtener una lista de objetos de tipo `Noticia` que posteriormente podremos manipular de la forma que queramos. Así de sencillo.

Tan sólo una anotación final. Para que este ejemplo funcione debemos añadir previamente permisos de acceso a internet para la aplicación. Esto se hace en el fichero `AndroidManifest.xml`, que quedaría de la siguiente forma:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="net.sgoliver"
      android:versionCode="1"
      android:versionName="1.0">

    <uses-permission
        android:name="android.permission.INTERNET" />

    <application android:icon="@drawable/icon"
                     android:label="@string/app_name">
        <activity android:name=".AndroidXml"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    </application>
    <uses-sdk android:minSdkVersion="7" />
</manifest>
```

En la línea 6 del código podéis ver cómo añadimos el permiso de acceso a la red mediante el elemento <uses-permission> con el parámetro android.permission.INTERNET

En los siguientes apartados veremos los otros dos métodos de tratamiento XML en Android que hemos comentado (DOM y StAX) y por último intentaremos comentar las diferencias entre ellos dependiendo del contexto de la aplicación.

> El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-xml-sax.zip

Tratamiento de XML con SAX Simplificado

En el apartado anterior del tutorial vimos cómo realizar la lectura y tratamiento de un documento XML utilizando el modelo SAX clásico. Vimos cómo implementar un *handler* SAX, donde se definían las acciones a realizar tras recibirse cada uno de los posibles eventos generados por el parser XML.

Este modelo, a pesar de funcionar perfectamente y de forma bastante eficiente, tiene claras desventajas. Por un lado se hace necesario definir una clase independiente para el handler. Adicionalmente, la naturaleza del modelo SAX implica la necesidad de poner bastante atención a la hora de definir dicho handler, ya que los eventos SAX definidos no están ligados de ninguna forma a etiquetas concretas del documento XML sino que se lanzarán para todas ellas, algo que obliga entre otras cosas a realizar la distinción entre etiquetas dentro de cada evento y a realizar otros chequeos adicionales.

Estos problemas se pueden observar perfectamente en el evento endElement() que definimos en el ejemplo del apartado anterior. En primer lugar teníamos que comprobar la condición de que el atributo noticiaActual no fuera null, para evitar confundir el elemento <title> descendiente de <channel> con el del mismo nombre pero descendiente de <item>. Posteriormente, teníamos que distinguir con un *IF* gigantesco entre todas las etiquetas posibles para realizar una acción u otra. Y todo esto para un

documento XML bastante sencillo. No es difícil darse cuenta de que para un documento XML algo más elaborado la complejidad del handler podría dispararse rápidamente, dando lugar a posibles errores.

Para evitar estos problemas, Android propone una variante del modelo SAX que evita definir una clase separada para el handler y que permite asociar directamente las acciones a etiquetas concretas dentro de la estructura del documento XML, lo que alivia en gran medida los inconvenientes mencionados.

Veamos cómo queda nuestro parser XML utilizando esta variante simplificada de SAX para Android y después comentaremos los aspectos más importantes del mismo.

```
import java.io.IOException;
import java.io.InputStream;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import org.xml.sax.Attributes;
import android.sax.Element;
import android.sax.EndElementListener;
import android.sax.EndTextElementListener;
import android.sax.RootElement;
import android.sax.StartElementListener;
import android.util.Xml;

public class RssParserSax2
{
    private URL rssUrl;
    private Noticia noticiaActual;

    public RssParserSax2(String url)
    {
        try
        {
            this.rssUrl = new URL(url);
        } catch (MalformedURLException e)
        {
            throw new RuntimeException(e);
        }
    }

    public List<Noticia> parse()
    {
        final List<Noticia> noticias = new ArrayList<Noticia>();

        RootElement root = new RootElement("rss");
        Element channel = root.getChild("channel");
        Element item = channel.getChild("item");

        item.setStartElementListener(new StartElementListener(){
            public void start(Attributes attrs) {
                noticiaActual = new Noticia();
            }
        });

        item.setEndElementListener(new EndElementListener(){
            public void end() {
                noticias.add(noticiaActual);
            }
        });

        item.getChild("title").setEndTextElementListener(
```

```
            new EndTextElementListener(){
                public void end(String body) {
                    noticiaActual.setTitulo(body);
                }
        });

        item.getChild("link").setEndTextElementListener(
            new EndTextElementListener(){
                public void end(String body) {
                    noticiaActual.setLink(body);
                }
        });

        item.getChild("description").setEndTextElementListener(
            new EndTextElementListener(){
                public void end(String body) {
                    noticiaActual.setDescripcion(body);
                }
        });

        item.getChild("guid").setEndTextElementListener(
            new EndTextElementListener(){
                public void end(String body) {
                    noticiaActual.setGuid(body);
                }
        });

        item.getChild("pubDate").setEndTextElementListener(
            new EndTextElementListener(){
                public void end(String body) {
                    noticiaActual.setFecha(body);
                }
        });

        try
        {
            Xml.parse(this.getInputStream(),
                    Xml.Encoding.UTF_8,
                    root.getContentHandler());
        } catch (Exception ex)
        {
            throw new RuntimeException(ex);
        }

        return noticias;
    }

    private InputStream getInputStream()
    {
        try
        {
            return rssUrl.openConnection().getInputStream();
        } catch (IOException e)
        {
            throw new RuntimeException(e);
        }
    }
}
```

Debemos atender principalmente al método `parse()`. En el modelo SAX clásico nos limitamos a instanciar al handler definido en una clase independiente y llamar al correspondiente método `parse()` de SAX. Por el contrario, en este nuevo modelo SAX simplificado de Android, las acciones a realizar para cada evento las vamos a definir en esta misma clase y además asociadas a etiquetas concretas del XML. Y para ello lo primero que

haremos será navegar por la estructura del XML hasta llegar a las etiquetas que nos interesa tratar y una vez allí, asignarle algunos de los *listeners* disponibles [de apertura (`StartElementListener`) o cierre (`EndElementListener`) de etiqueta] incluyendo las acciones oportunas. De esta forma, para el elemento `<item>` navegaremos hasta él obteniendo en primer lugar el elemento raíz del XML (`<rss>`) declarando un nuevo objeto `RootElement` y después accederemos a su elemento hijo `<channel>` y a su vez a su elemento hijo `<item>`, utilizando en cada paso el método `getChild()`. Una vez hemos llegado a la etiqueta deseada, asignaremos los listeners necesarios, en nuestro caso uno de apertura de etiqueta y otro de cierre, donde inicializaremos la noticia actual y la añadiremos a la lista final respectivamente, de forma análoga a lo que hacíamos para el modelo SAX clásico. Para el resto de etiquetas actuaremos de la misma forma, accediendo a ellas con `getChild()` y asignado los listeners necesarios.

Finalmente, iniciaremos el proceso de parsing simplemente llamando al método `parse()` definido en la clase `android.Util.Xml`, al que pasaremos como parámetros el stream de entrada, la codificación del documento XML y un handler SAX obtenido directamente del objeto `RootElement` definido anteriormente.

Como vemos, este modelo SAX alternativo simplifica la elaboración del handler necesario y puede ayudar a evitar posibles errores en el handler y disminuir la complejidad del mismo para casos en los que el documento XML no sea tan sencillo como el utilizado para estos ejemplos. Por supuesto, el modelo clásico es tan válido y eficiente como éste, por lo que la elección entre ambos es cuestión de gustos.

En el siguiente apartado pasaremos ya a describir el siguiente de los métodos de lectura de XML en Android, llamado *Document Object Model* (DOM).

> El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-xml-sax2.zip

Tratamiento de XML con DOM

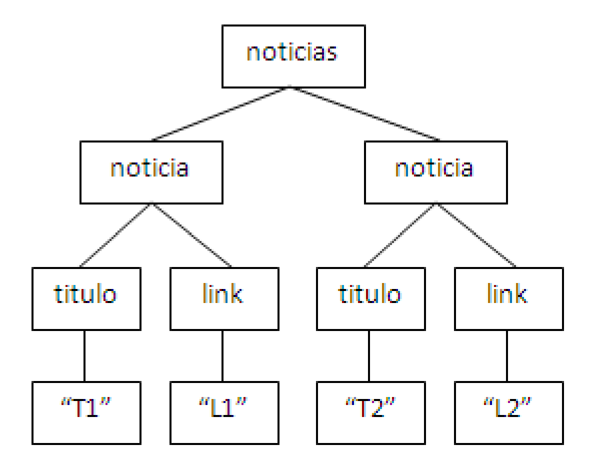
En el apartado anterior del curso de programación para Android hablamos sobre SAX, el primero de los métodos disponibles en Android para leer ficheros XML desde nuestras aplicaciones. En este segundo apartado vamos a centrarnos en DOM, otro de los métodos clásicos para la lectura y tratamiento de XML.

Cuando comentábamos la filosofía de SAX ya vimos cómo con dicho modelo el tratamiento del fichero XML se realizaba de forma secuencial, es decir, se iban realizando las acciones necesarias durante la propia lectura del documento. Sin embargo, con DOM la estrategia cambia radicalmente. Con DOM, el documento XML se lee completamente antes de poder realizar ninguna acción en función de su contenido. Esto es posible gracias a que, como resultado de la lectura del documento, el parser DOM devuelve todo su contenido en forma de una estructura de tipo árbol, donde los distintos elementos del XML se representa en forma de nodos y su jerarquía padre-hijo se establece mediante relaciones entre dichos nodos.

Como ejemplo, vemos un ejemplo de XML sencillo y cómo quedaría su representación en forma de árbol:

```
<noticias>
    <noticia>
        <titulo>T1</titulo>
        <link>L1</link>
    </noticia>
    <noticia>
        <titulo>T2</titulo>
        <link>L2</link>
    </noticia>
<noticias>
```

Este XML se traduciría en un árbol parecido al siguiente:



Como vemos, este árbol conserva la misma información contenida en el fichero XML pero en forma de nodos y transiciones entre nodos, de forma que se puede navegar fácilmente por la estructura. Además, este árbol se conserva persistente en memoria una vez leído el documento completo, lo que permite procesarlo en cualquier orden y tantas veces como sea necesario (a diferencia de SAX, donde el tratamiento era secuencial y siempre de principio a fin del documento, no pudiendo volver atrás una vez finalizada la lectura del XML).

Para todo esto, el modelo DOM ofrece una serie de clases y métodos que permiten almacenar la información de la forma descrita y facilitan la navegación y el tratamiento de la estructura creada.

Veamos cómo quedaría nuestro parser utilizando el modelo DOM y justo después comentaremos los detalles más importantes.

```
public class RssParserDom
{
    private URL rssUrl;

    public RssParserDom(String url)
    {
        try
        {
            this.rssUrl = new URL(url);
        } catch (MalformedURLException e)
```

```
        {
            throw new RuntimeException(e);
        }
    }

    public List<Noticia> parse()
    {
        //Instanciamos la fábrica para DOM
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        List<Noticia> noticias = new ArrayList<Noticia>();

        try
        {
            //Creamos un nuevo parser DOM
            DocumentBuilder builder = factory.newDocumentBuilder();

            //Realizamos lalectura completa del XML
            Document dom = builder.parse(this.getInputStream());

            //Nos posicionamos en el nodo principal del árbol (<rss>)
            Element root = dom.getDocumentElement();

            //Localizamos todos los elementos <item>
            NodeList items = root.getElementsByTagName("item");

            //Recorremos la lista de noticias
            for (int i=0; i<items.getLength(); i++)
            {
                Noticia noticia = new Noticia();

                //Obtenemos la noticia actual
                Node item = items.item(i);

                //Obtenemos la lista de datos de la noticia actual
                NodeList datosNoticia = item.getChildNodes();

                //Procesamos cada dato de la noticia
                for (int j=0; j<datosNoticia.getLength(); j++)
                {
                    Node dato = datosNoticia.item(j);
                    String etiqueta = dato.getNodeName();

                    if (etiqueta.equals("title"))
                    {
                        String texto = obtenerTexto(dato);
                        noticia.setTitulo(texto);
                    }
                    else if (etiqueta.equals("link"))
                    {
                        noticia.setLink(dato.getFirstChild().getNodeValue());
                    }
                    else if (etiqueta.equals("description"))
                    {
                        String texto = obtenerTexto(dato);
                        noticia.setDescripcion(texto);
                    }
                    else if (etiqueta.equals("guid"))
                    {
                        noticia.setGuid(dato.getFirstChild().getNodeValue());
                    }
                    else if (etiqueta.equals("pubDate"))
                    {
                        noticia.setFecha(dato.getFirstChild().getNodeValue());
                    }
                }

                noticias.add(noticia);
            }
```

```
        }
        catch (Exception ex)
        {
            throw new RuntimeException(ex);
        }

        return noticias;
    }

    private String obtenerTexto(Node dato)
    {
        StringBuilder texto = new StringBuilder();
        NodeList fragmentos = dato.getChildNodes();

        for (int k=0;k<fragmentos.getLength();k++)
        {
            texto.append(fragmentos.item(k).getNodeValue());
        }

        return texto.toString();
    }

    private InputStream getInputStream()
    {
        try
        {
            return rssUrl.openConnection().getInputStream();
        }
        catch (IOException e)
        {
            throw new RuntimeException(e);
        }
    }
}
```

Nos centramos una vez más en el método `parse()`. Al igual que hacíamos para SAX, el primer paso será instanciar una nueva fábrica, esta vez de tipo `DocumentBuilderFactory`, y posteriormente crear un nuevo parser a partir de ella mediante el método `newDocumentBuilder()`.

Tras esto, ya podemos realizar la lectura del documento XML llamando al método `parse()` de nuestro parser DOM, pasándole como parámetro el *stream* de entrada del fichero. Al hacer esto, el documento XML se leerá completo y se generará la estructura de árbol equivalente, que se devolverá como un objeto de tipo `Document`. Éste será el objeto que podremos navegar para realizar el tratamiento necesario del XML.

Para ello, lo primero que haremos será acceder al nodo principal del árbol (en nuestro caso, la etiqueta `<rss>`) utilizando el método `getDocumentElement()`. Una vez posicionados en dicho nodo, vamos a buscar todos los nodos cuya etiqueta sea `<item>`. Esto lo conseguimos utilizando el método de búsqueda por nombre de etiqueta, `getElementsByTagName("nombre_de_etiqueta")`, que devolverá una lista (de tipo `NodeList`) con todos los nodos hijos del nodo actual cuya etiqueta coincida con la pasada como parámetro.

Una vez tenemos localizados todos los elementos `<item>`, que representan a cada noticia, los vamos a recorrer uno a uno para ir generando todos los objetos `Noticia` necesarios. Para cada uno de ellos, se obtendrán los nodos hijos del elemento mediante `getChildNodes()`

y se recorrerán éstos obteniendo su texto y almacenándolo en el atributo correspondiente del objeto Noticia. Para saber a qué etiqueta corresponde cada nodo hijo utilizamos el método getNodeName().

Merece la pena pararnos un poco en comentar la forma de obtener el texto contenido en un nodo. Como vimos al principio del apartado en el ejemplo gráfico de árbol DOM, el texto de un nodo determinado se almacena a su vez como nodo hijo de dicho nodo. Este nodo de texto suele ser único, por lo que la forma habitual de obtener el texto de un nodo es obtener su primer nodo hijo y de éste último obtener su valor:

```
String texto = nodo.getFirstChild().getNodeValue();
```

Sin embargo, en ocasiones, el texto contenido en el nodo viene fragmentado en varios nodos hijos, en vez de sólo uno. Esto ocurre por ejemplo cuando se utilizan en el texto entidades HTML, como por ejemplo " . En estas ocasiones, para obtener el texto completo hay que recorrer todos los nodos hijos e ir concatenando el texto de cada uno para formar el texto completo.

Esto es lo que hace nuestra función auxiliar obtenerTexto():

```
private String obtenerTexto(Node dato)
{
        StringBuilder texto = new StringBuilder();
        NodeList fragmentos = dato.getChildNodes();

        for (int k=0;k<fragmentos.getLength();k++)
        {
            texto.append(fragmentos.item(k).getNodeValue());
        }

        return texto.toString();
}
```

Como vemos, el modelo DOM nos permite localizar y tratar determinados elementos concretos del documento XML, sin la necesidad de recorrer todo su contenido de principio a fin. Además, a diferencia de SAX, como tenemos cargado en memoria el documento completo de forma persistente (en forma de objeto Document), podremos consultar, recorrer y tratar el documento tantas veces como sea necesario sin necesidad de volverlo a parsear. En un apartado posterior veremos como todas estas características pueden ser ventajas o inconvenientes según el contexto de la aplicación y el tipo de XML tratado.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-xml-dom.zip

Tratamiento de XML con XmlPull

En los apartados anteriores dedicados al tratamiento de XML en aplicaciones Android dentro de nuestro tutorial de programación Android hemos comentado ya los modelos SAX y DOM, los dos métodos más comunes de lectura de XML soportados en la plataforma.

En este cuarto apartado nos vamos a centrar en el último método menos conocido, aunque igual de válido según el contexto de la aplicación, llamado *XmlPull*. Este método es una versión similar al modelo StAX (*Streaming API for XML*), que en esencia es muy parecido al modelo SAX ya comentado. Y digo muy parecido porque también se basa en definir las acciones a realizar para cada uno de los eventos generados durante la lectura secuencial del documento XML. ¿Cuál es la diferencia entonces?

La diferencia radica principalmente en que, mientras que en SAX no teníamos control sobre la lectura del XML una vez iniciada (el parser lee automáticamente el XML de principio a fin generando todos los eventos necesarios), en el modelo XmlPull vamos a poder guiar o intervenir en la lectura del documento, siendo nosotros los que vayamos pidiendo de forma explícita la lectura del siguiente elemento del XML y respondiendo al resultado ejecutando las acciones oportunas.

Veamos cómo podemos hacer esto:

```
public class RssParserPull
{
    private URL rssUrl;

    public RssParserPull(String url)
    {
        try
        {
            this.rssUrl = new URL(url);
        }
        catch (MalformedURLException e)
        {
            throw new RuntimeException(e);
        }
    }

    public List<Noticia> parse()
    {
        List<Noticia> noticias = null;
        XmlPullParser parser = Xml.newPullParser();

        try
        {
            parser.setInput(this.getInputStream(), null);

            int evento = parser.getEventType();

            Noticia noticiaActual = null;

            while (evento != XmlPullParser.END_DOCUMENT)
            {
                String etiqueta = null;


                switch (evento)
```

```
            {
                case XmlPullParser.START_DOCUMENT:

                    noticias = new ArrayList<Noticia>();
                    break;

                case XmlPullParser.START_TAG:

                    etiqueta = parser.getName();

                    if (etiqueta.equals("item"))
                    {
                        noticiaActual = new Noticia();
                    }
                    else if (noticiaActual != null)
                    {
                        if (etiqueta.equals("link"))
                        {
                            noticiaActual.setLink(parser.nextText());
                        }
                        else if (etiqueta.equals("description"))
                        {
                            noticiaActual.setDescripcion(
                                                 parser.nextText());
                        }
                        else if (etiqueta.equals("pubDate"))
                        {
                            noticiaActual.setFecha(parser.nextText());
                        }
                        else if (etiqueta.equals("title"))
                        {
                            noticiaActual.setTitulo(parser.nextText());
                        }
                        else if (etiqueta.equals("guid"))
                        {
                            noticiaActual.setGuid(parser.nextText());
                        }
                    }
                    break;

                case XmlPullParser.END_TAG:

                    etiqueta = parser.getName();

                    if (etiqueta.equals("item") && noticiaActual != null)
                    {
                        noticias.add(noticiaActual);
                    }
                    break;
            }

            evento = parser.next();
        }
    }
    catch (Exception ex)
    {
        throw new RuntimeException(ex);
    }

    return noticias;
}




private InputStream getInputStream()
{
```

```
        try
        {
            return rssUrl.openConnection().getInputStream();
        }
        catch (IOException e)
        {
            throw new RuntimeException(e);
        }
    }
}
```

Centrándonos una vez más en el método `parse()`, vemos que tras crear el nuevo parser XmlPull y establecer el fichero de entrada en forma de *stream* [mediante `XmlPull.newPullParser()` y `parser.setInput(...)`] nos metemos en un bucle en el que iremos solicitando al parser en cada paso el siguiente evento encontrado en la lectura del XML, utilizando para ello el método `parser.next()`. Para cada evento devuelto como resultado consultaremos su tipo mediante el método `parser.getEventType()` y responderemos con las acciones oportunas según dicho tipo (`START_DOCUMENT`, `END_DOCUMENT`, `START_TAG`, `END_TAG`). Una vez identificado el tipo concreto de evento, podremos consultar el nombre de la etiqueta del elemento XML mediante `parser.getName()` y su texto correspondiente mediante `parser.nextText()`. En cuanto a la obtención del texto, con este modelo tenemos la ventaja de no tener que preocuparnos por "recolectar" todos los fragmentos de texto contenidos en el elemento XML, ya que `nextText()` devolverá todo el texto que encuentre hasta el próximo evento de fin de etiqueta (`ENT_TAG`).

Y sobre este modelo de tratamiento no queda mucho más que decir, ya que las acciones ejecutadas en cada caso son análogas a las que ya vimos en los apartados anteriores.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-xml-pull.zip



Bases de Datos

VI. Bases de Datos

Primeros pasos con SQLite

En los siguientes apartados de este tutorial de programación Android, nos vamos a detener en describir las distintas opciones de acceso a datos que proporciona la plataforma y en cómo podemos realizar las tareas más habituales dentro de este apartado.

La plataforma Android proporciona dos herramientas principales para el almacenamiento y consulta de datos estructurados:

- Bases de Datos SQLite
- Content Providers

En estos próximos apartados nos centraremos en la primera opción, SQLite, que abarcará todas las tareas relacionadas con el almacenamiento de los datos propios de nuestra aplicación. El segundo de los mecanismos, los *Content Providers*, que trataremos más adelante, nos facilitarán la tarea de hacer visibles esos datos a otras aplicaciones y, de forma recíproca, de permitir la consulta de datos publicados por terceros desde nuestra aplicación.

SQLite es un motor de bases de datos muy popular en la actualidad por ofrecer características tan interesantes como su pequeño tamaño, no necesitar servidor, precisar poca configuración, ser transaccional y por supuesto ser de código libre.

Android incorpora de serie todas las herramientas necesarias para la creación y gestión de bases de datos SQLite, y entre ellas una completa API para llevar a cabo de manera sencilla todas las tareas necesarias. Sin embargo, en este primer apartado sobre bases de datos en Android no vamos a entrar en mucho detalle con esta API. Por el momento nos limitaremos a ver el código necesario para crear una base de datos, insertaremos algún dato de prueba, y veremos cómo podemos comprobar que todo funciona correctamente.

En Android, la forma típica para crear, actualizar, y conectar con una base de datos SQLite será a través de una clase auxiliar llamada `SQLiteOpenHelper`, o para ser más exactos, de una clase propia que derive de ella y que debemos personalizar para adaptarnos a las necesidades concretas de nuestra aplicación.

La clase `SQLiteOpenHelper` tiene tan sólo un constructor, que normalmente no necesitaremos sobrescribir, y dos métodos abstractos, `onCreate()` y `onUpgrade()`, que deberemos personalizar con el código necesario para crear nuestra base de datos y para actualizar su estructura respectivamente.

Como ejemplo, nosotros vamos a crear una base de datos muy sencilla llamada `BDUsuarios`, con una sola tabla llamada Usuarios que contendrá sólo dos campos: nombre e email. Para ellos, vamos a crear una clase derivada de `SQLiteOpenHelper` que llamaremos `UsuariosSQLiteHelper`, donde sobrescribiremos los métodos `onCreate()` y `onUpgrade()` para adaptarlos a la estructura de datos indicada:

```
package net.sgoliver.android;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.database.sqlite.SQLiteOpenHelper;

public class UsuariosSQLiteHelper extends SQLiteOpenHelper {

    //Sentencia SQL para crear la tabla de Usuarios
    String sqlCreate = "CREATE TABLE Usuarios (codigo INTEGER, nombre TEXT)";

    public UsuariosSQLiteHelper(Context contexto, String nombre,
                               CursorFactory factory, int version) {
        super(contexto, nombre, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        //Se ejecuta la sentencia SQL de creación de la tabla
        db.execSQL(sqlCreate);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int versionAnterior,
                          int versionNueva) {
        //NOTA: Por simplicidad del ejemplo aquí utilizamos directamente
        //      la opción de eliminar la tabla anterior y crearla de nuevo
        //      vacía con el nuevo formato.
        //      Sin embargo lo normal será que haya que migrar datos de la
        //      tabla antigua a la nueva, por lo que este método debería
        //      ser más elaborado.

        //Se elimina la versión anterior de la tabla
        db.execSQL("DROP TABLE IF EXISTS Usuarios");

        //Se crea la nueva versión de la tabla
        db.execSQL(sqlCreate);
    }
}
```

Lo primero que hacemos es definir una variable llamado `sqlCreate` donde almacenamos la sentencia SQL para crear una tabla llamada Usuarios con los campos alfanuméricos nombre e email. NOTA: No es objetivo de este tutorial describir la sintaxis del lenguaje SQL ni las particularidades del motor de base de datos SQLite, por lo que no entraré a describir las sentencias SQL utilizadas. Para más información sobre SQLite puedes consultar la documentación oficial o empezar por leer una pequeña introducción que hice en este mismo blog cuando traté el tema de utilizar SQLite desde aplicaciones .NET

El método `onCreate()` será ejecutado automáticamente por nuestra clase `UsuariosDBHelper` cuando sea necesaria la creación de la base de datos, es decir, cuando aún no exista. Las tareas típicas que deben hacerse en este método serán la creación de todas las tablas necesarias y la inserción de los datos iniciales si son necesarios. En nuestro caso, sólo vamos a crear la tabla Usuarios descrita anteriormente. Para la creación de la tabla utilizaremos la sentencia SQL ya definida y la ejecutaremos contra la base de datos utilizando el método más sencillo de los disponibles en la API de SQLite proporcionada por Android, llamado `execSQL()`. Este método se limita a ejecutar directamente el código SQL que le pasemos como parámetro.

Por su parte, el método onUpgrade() se lanzará automáticamente cuando sea necesaria una actualización de la estructura de la base de datos o una conversión de los datos.

Un ejemplo práctico: imaginemos que publicamos una aplicación que utiliza una tabla con los campos usuario e email (llamémoslo versión 1 de la base de datos). Más adelante, ampliamos la funcionalidad de nuestra aplicación y necesitamos que la tabla también incluya un campo adicional por ejemplo con la edad del usuario (versión 2 de nuestra base de datos). Pues bien, para que todo funcione correctamente, la primera vez que ejecutemos la versión ampliada de la aplicación necesitaremos modificar la estructura de la tabla Usuarios para añadir el nuevo campo edad. Pues este tipo de cosas son las que se encargará de hacer automáticamente el método onUpgrade() cuando intentemos abrir una versión concreta de la base de datos que aún no exista. Para ello, como parámetros recibe la versión actual de la base de datos en el sistema, y la nueva versión a la que se quiere convertir. En función de esta pareja de datos necesitaremos realizar unas acciones u otras. En nuestro caso de ejemplo optamos por la opción más sencilla: borrar la tabla actual y volver a crearla con la nueva estructura, pero como se indica en los comentarios del código, lo habitual será que necesitemos algo más de lógica para convertir la base de datos de una versión a otra y por supuesto para conservar los datos registrados hasta el momento.

Una vez definida nuestra clase *helper*, la apertura de la base de datos desde nuestra aplicación resulta ser algo de lo más sencillo. Lo primero será crear un objeto de la clase UsuariosSQLiteHelper al que pasaremos el contexto de la aplicación (en el ejemplo una referencia a la actividad principal), el nombre de la base de datos, un objeto CursorFactory que típicamente no será necesario (en ese caso pasaremos el valor null), y por último la versión de la base de datos que necesitamos. La simple creación de este objeto puede tener varios efectos:

- Si la base de datos ya existe y su versión actual coincide con la solicitada simplemente se realizará la conexión con ella.
- Si la base de datos existe pero su versión actual es anterior a la solicitada, se llamará automáticamente al método onUpgrade() para convertir la base de datos a la nueva versión y se conectará con la base de datos convertida.
- Si la base de datos no existe, se llamará automáticamente al método onCreate() para crearla y se conectará con la base de datos creada.

Una vez tenemos una referencia al objeto UsuariosSQLiteHelper, llamaremos a su método getReadableDatabase() o getWritableDatabase() para obtener una referencia a la base de datos, dependiendo si sólo necesitamos consultar los datos o también necesitamos realizar modificaciones, respectivamente.

Ahora que ya hemos conseguido una referencia a la base de datos (objeto de tipo SQLiteDatabase) ya podemos realizar todas las acciones que queramos sobre ella. Para nuestro ejemplo nos limitaremos a insertar 5 registros de prueba, utilizando para ello el método ya comentado execSQL() con las sentencias INSERT correspondientes. Por último cerramos la conexión con la base de datos llamando al método close().

```
package net.sgoliver.android;

import android.app.Activity;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;

public class AndroidBaseDatos extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        //Abrimos la base de datos 'DBUsuarios' en modo escritura
        UsuariosSQLiteHelper usdbh =
            new UsuariosSQLiteHelper(this, "DBUsuarios", null, 1);

        SQLiteDatabase db = usdbh.getWritableDatabase();

        //Si hemos abierto correctamente la base de datos
        if(db != null)
        {
            //Insertamos 5 usuarios de ejemplo
            for(int i=1; i<=5; i++)
            {
                //Generamos los datos
                int codigo = i;
                String nombre = "Usuario" + i;

                //Insertamos los datos en la tabla Usuarios
                db.execSQL("INSERT INTO Usuarios (codigo, nombre) " +
                           "VALUES (" + codigo + ", '" + nombre +"')");
            }

            //Cerramos la base de datos
            db.close();
        }
    }
}
```

Vale, ¿y ahora qué? ¿dónde está la base de datos que acabamos de crear? ¿cómo podemos comprobar que todo ha ido bien y que los registros se han insertado correctamente? Vayamos por partes.

En primer lugar veamos dónde se ha creado nuestra base de datos. Todas las bases de datos SQLite creadas por aplicaciones Android se almacenan en la memoria del teléfono en un fichero con el mismo nombre de la base de datos situado en una ruta que sigue el siguiente patrón:

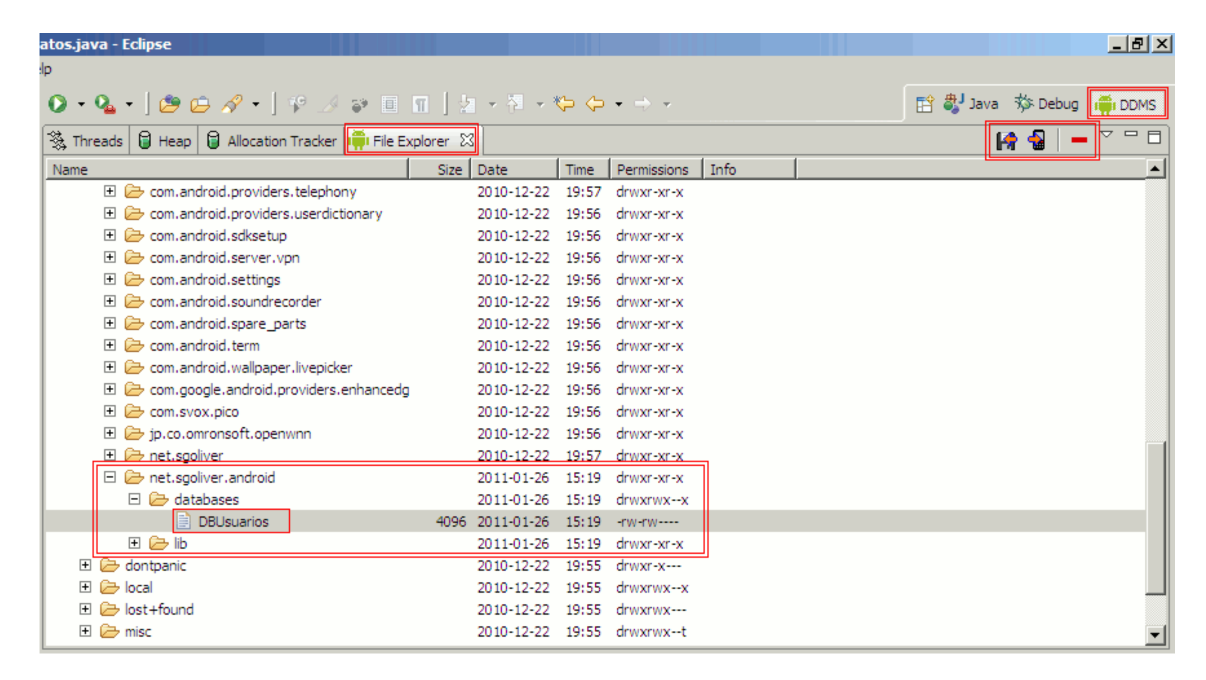
`/data/data/paquete.java.de.la.aplicacion/databases/nombre_base_datos`

En el caso de nuestro ejemplo, la base de datos se almacenaría por tanto en la ruta siguiente:

`/data/data/net.sgoliver.android/databases/DBUsuarios`

Para comprobar esto podemos hacer lo siguiente. Una vez ejecutada por primera vez desde Eclipse la aplicación de ejemplo sobre el emulador de Android (y por supuesto antes de cerrarlo) podemos ir a la perspectiva "DDMS" (*Dalvik Debug Monitor Server*) de Eclipse y en la solapa "*File Explorer*" podremos acceder al sistema de archivos del emulador, donde

podremos buscar la ruta indicada de la base de datos. Podemos ver esto en la siguiente imagen:



Con esto ya comprobamos al menos que el fichero de nuestra base de datos se ha creado en la ruta correcta. Ya sólo nos queda comprobar que tanto las tablas creadas como los datos insertados también se han incluido correctamente en la base de datos. Para ello podemos recurrir a dos posibles métodos:

1. Transferir la base de datos a nuestro PC y consultarla con cualquier administrador de bases de datos SQLite.
2. Acceder directamente a la consola de comandos del emulador de Android y utilizar los comandos existentes para acceder y consultar la base de datos SQLite.

El primero de los métodos es sencillo. El fichero de la base de datos podemos transferirlo a nuestro PC utilizando el botón de descarga situado en la esquina superior derecha del explorador de archivos (remarcado en rojo en la imagen anterior). Junto a este botón aparecen otros dos para hacer la operación contraria (copiar un fichero local al sistema de archivos del emulador) y para eliminar ficheros del emulador. Una vez descargado el fichero a nuestro sistema local, podemos utilizar cualquier administrador de SQLite para abrir y consultar la base de datos, por ejemplo SQLite Administrator (freeware).

El segundo método utiliza una estrategia diferente. En vez de descargar la base de datos a nuestro sistema local, somos nosotros los que accedemos de forma remota al emulador a través de su consola de comandos (*shell*). Para ello, con el emulador de Android aún abierto, debemos abrir una consola de MS-DOS y utilizar la utilidad adb.exe (*Android Debug Bridge*) situada en la carpeta `platform-tools` del SDK de Android (en mi caso: `c:\android-sdk-windows\platform-tools\`). En primer lugar consultaremos los identificadores de todos los emuladores en ejecución mediante el comando "`adb devices`". Esto nos debe devolver una única instancia si sólo tenemos un emulador abierto, que en mi caso particular se llama "`emulator-5554`".

Tras conocer el identificador de nuestro emulador, vamos a acceder a su shell mediante el comando "`adb -s identificador-del-emulador shell`". Una vez conectados, ya podemos acceder a nuestra base de datos utilizando el comando `sqlite3` pasándole la ruta del fichero, para nuestro ejemplo "`sqlite3 /data/data/net.sgoliver.android/databases/DBUsuarios`". Si todo ha ido bien, debe aparecernos el *prompt* de SQLite "`sqlite>`", lo que nos indicará que ya podemos escribir las consultas SQL necesarias sobre nuestra base de datos. Nosotros vamos a comprobar que existe la tabla Usuarios y que se han insertado los cinco registros de ejemplo. Para ello haremos la siguiente consulta: "`SELECT * FROM Usuarios;`". Si todo es correcto esta instrucción debe devolvernos los cinco usuarios existentes en la tabla. En la imagen siguiente se muestra todo el proceso descrito:

```
C:\WINDOWS\system32\cmd.exe

C:\android-sdk-windows\platform-tools>adb devices
List of devices attached
emulator-5554   device


C:\android-sdk-windows\platform-tools>adb -s emulator-5554 shell
# sqlite3 /data/data/net.sgoliver.android/databases/DBUsuarios
sqlite3 /data/data/net.sgoliver.android/databases/DBUsuarios
SQLite version 3.5.9
Enter ".help" for instructions
sqlite> select * from Usuarios;
select * from Usuarios;
1|Usuario1
2|Usuario2
3|Usuario3
4|Usuario4
5|Usuario5
sqlite> .exit
.exit
# exit
exit

C:\android-sdk-windows\platform-tools>_
```

Con esto ya hemos comprobado que nuestra base de datos se ha creado correctamente, que se han insertado todos los registros de ejemplo y que todo funciona según se espera.

En los siguientes apartados comentaremos las distintas posibilidades que tenemos a la hora de manipular los datos de la base de datos (insertar, eliminar y modificar datos) y cómo podemos realizar consultas sobre los mismos, ya que [como siempre] tendremos varias opciones disponibles.

Insertar/Actualizar/Eliminar

En el apartado anterior del curso de programación en Android vimos cómo crear una base de datos para utilizarla desde nuestra aplicación Android. En este segundo apartado de la serie vamos a describir las posibles alternativas que proporciona la API de Android a la hora de insertar, actualizar y eliminar registros de nuestra base de datos SQLite.

La API de SQLite de Android proporciona dos alternativas para realizar operaciones sobre la base de datos que no devuelven resultados (entre ellas la inserción/actualización/eliminación de registros, pero también la creación de tablas, de índices, etc).

El primero de ellos, que ya comentamos brevemente en el apartado anterior, es el método `execSQL()` de la clase `SQLiteDatabase`. Este método permite ejecutar cualquier sentencia SQL sobre la base de datos, siempre que ésta no devuelva resultados. Para ello, simplemente aportaremos como parámetro de entrada de este método la cadena de texto correspondiente con la sentencia SQL. Cuando creamos la base de datos en el post anterior ya vimos algún ejemplo de esto para insertar los registros de prueba. Otros ejemplos podrían ser los siguientes:

```
//Insertar un registro
db.execSQL("INSERT INTO Usuarios (usuario,email) VALUES
('usu1','usu1@email.com') ");

//Eliminar un registro
db.execSQL("DELETE FROM Usuarios WHERE usuario='usu1' ");

//Actualizar un registro
db.execSQL("UPDATE Usuarios SET email='nuevo@email.com' WHERE usuario='usu1'
");
```

La segunda de las alternativas disponibles en la API de Android es utilizar los métodos `insert()`, `update()` y `delete()` proporcionados también con la clase `SQLiteDatabase`. Estos métodos permiten realizar las tareas de inserción, actualización y eliminación de registros de una forma algo más paramétrica que `execSQL()`, separando tablas, valores y condiciones en parámetros independientes de estos métodos.

Empecemos por el método `insert()` para insertar nuevos registros en la base de datos. Este método recibe tres parámetros, el primero de ellos será el nombre de la tabla, el tercero serán los valores del registro a insertar, y el segundo lo obviaremos por el momento ya que tan sólo se hace necesario en casos muy puntuales (por ejemplo para poder insertar registros completamente vacíos), en cualquier otro caso pasaremos con valor `null` este segundo parámetro.

Los valores a insertar los pasaremos como elementos de una colección de tipo `ContentValues`. Esta colección es de tipo diccionario, donde almacenaremos parejas de clave-valor, donde la clave será el nombre de cada campo y el valor será el dato correspondiente a insertar en dicho campo.

Veamos un ejemplo:

```
//Creamos el registro a insertar como objeto ContentValues
ContentValues nuevoRegistro = new ContentValues();
nuevoRegistro.put("usuario", "usu10");
nuevoRegistro.put("email","usu10@email.com");

//Insertamos el registro en la base de datos
db.insert("Usuarios", null, nuevoRegistro);
```

Los métodos `update()` y `delete()` se utilizarán de forma muy parecida a ésta, con la salvedad de que recibirán un parámetro adicional con la condición `WHERE` de la sentencia SQL. Por ejemplo, para actualizar el email del usuario de nombre 'usu1' haríamos lo siguiente:

```
//Establecemos los campos-valores a actualizar
ContentValues valores = new ContentValues();
valores.put("email","usu1_nuevo@email.com");

//Actualizamos el registro en la base de datos
db.update("Usuarios", valores, "usuario='usu1'");
```

Como podemos ver, como tercer parámetro del método `update()` pasamos directamente la condición del `UPDATE` tal como lo haríamos en la cláusula `WHERE` en una sentencia SQL normal.

El método `delete()` se utilizaría de forma análoga. Por ejemplo para eliminar el registro del usuario 'usu2' haríamos lo siguiente:

```
//Eliminamos el registro del usuario 'usu2'
db.delete("Usuarios", "usuario='usu2'");
```

Como vemos, volvemos a pasar como primer parámetro el nombre de la tabla y en segundo lugar la condición `WHERE`. Por supuesto, si no necesitáramos ninguna condición, podríamos dejar como `null` en este parámetro.

Un último detalle sobre estos métodos. Tanto en el caso de `execSQL()` como en los casos de `update()` o `delete()` podemos utilizar argumentos dentro de las condiciones de la sentencia SQL. Esto no son más que partes variables de la sentencia SQL que aportaremos en un *array* de valores aparte, lo que nos evitará pasar por la situación típica en la que tenemos que construir una sentencia SQL concatenando cadenas de texto y variables para formar el comando SQL final. Estos argumentos SQL se indicarán con el símbolo '`?`', y los valores de dichos argumentos deben pasarse en el array en el mismo orden que aparecen en la sentencia SQL. Así, por ejemplo, podemos escribir instrucciones como la siguiente:

```
//Eliminar un registro con execSQL(), utilizando argumentos
String[] args = new String[]{"usu1"};
db.execSQL("DELETE FROM Usuarios WHERE usuario=?", args);

//Actualizar dos registros con update(), utilizando argumentos
ContentValues valores = new ContentValues();
valores.put("email","usu1_nuevo@email.com");

String[] args = new String[]{"usu1", "usu2"};
db.update("Usuarios", valores, "usuario=? OR usuario=?", args);
```

Esta forma de pasar a la sentencia SQL determinados datos variables puede ayudarnos además a escribir código más limpio y evitar posibles errores.

En el siguiente apartado veremos cómo consultar la base de datos para recuperar registros según un determinado criterio.

Consultar/Recuperar registros

En el anterior apartado del curso vimos todas las opciones disponibles a la hora de insertar, actualizar y eliminar datos de una base de datos SQLite en Android. En esta nueva entrega

vamos a describir la última de las tareas importantes de tratamiento de datos que nos queda por ver, la selección y recuperación de datos.

De forma análoga a lo que vimos para las sentencias de modificación de datos, vamos a tener dos opciones principales para recuperar registros de una base de datos SQLite en Android. La primera de ellas utilizando directamente un comando de selección SQL, y como segunda opción utilizando un método específico donde parametrizaremos la consulta a la base de datos.

Para la primera opción utilizaremos el método `rawQuery()` de la clase `SQLiteDatabase`. Este método recibe directamente como parámetro un comando SQL completo, donde indicamos los campos a recuperar y los criterios de selección. El resultado de la consulta lo obtendremos en forma de cursor, que posteriormente podremos recorrer para procesar los registros recuperados. Sirva la siguiente consulta a modo de ejemplo:

```
Cursor c = db.rawQuery(" SELECT usuario,email FROM Usuarios WHERE
usuario='usu1' ");
```

Como en el caso de los métodos de modificación de datos, también podemos añadir a este método una lista de argumentos variables que hayamos indicado en el comando SQL con el símbolo '`?`', por ejemplo así:

```
String[] args = new String[] {"usu1"};
Cursor c = db.rawQuery(" SELECT usuario,email FROM Usuarios WHERE usuario=? ",
args);
```

Más adelante en este apartado veremos cómo podemos manipular el objeto Cursor para recuperar los datos obtenidos.

Como segunda opción para recuperar datos podemos utilizar el método `query()` de la clase `SQLiteDatabase`. Este método recibe varios parámetros: el nombre de la tabla, un array con los nombre de campos a recuperar, la cláusula `WHERE`, un array con los argumentos variables incluidos en el `WHERE` (si los hay, `null` en caso contrario), la cláusula GROUP BY si existe, la cláusula `HAVING` si existe, y por último la cláusula `ORDER BY` si existe. Opcionalmente, se puede incluir un parámetro al final más indicando el número máximo de registros que queremos que nos devuelva la consulta. Veamos el mismo ejemplo anterior utilizando el método `query()`:

```
String[] campos = new String[] {"usuario", "email"};
String[] args = new String[] {"usu1"};

Cursor c = db.query("Usuarios", campos, "usuario=?", args, null, null, null);
```

Como vemos, los resultados se devuelven nuevamente en un objeto Cursor que deberemos recorrer para procesar los datos obtenidos.

Para recorrer y manipular el cursor devuelto por cualquiera de los dos métodos mencionados tenemos a nuestra disposición varios métodos de la clase Cursor, entre los que destacamos dos de los dedicados a recorrer el cursor de forma secuencial y en orden natural:

- moveToFirst(): mueve el puntero del cursor al primer registro devuelto.
- moveToNext(): mueve el puntero del cursor al siguiente registro devuelto.

Los métodos moveToFirst() y moveToNext() devuelven TRUE en caso de haber realizado el movimiento correspondiente del puntero sin errores, es decir, siempre que exista un primer registro o un registro siguiente, respectivamente.

Una vez posicionados en cada registro podremos utilizar cualquiera de los métodos getXXX(índice_columna) existentes para cada tipo de dato para recuperar el dato de cada campo del registro actual del cursor. Así, si queremos recuperar por ejemplo la segunda columna del registro actual, y ésta contiene un campo alfanumérico, haremos la llamada getString(1) [NOTA: los índices comienzan por 0, por lo que la segunda columna tiene índice 1], en caso de contener un dato de tipo real llamaríamos a getDouble(1), y de forma análoga para todos los tipos de datos existentes. Con todo esto en cuenta, veamos cómo podríamos recorrer el cursor devuelto por el ejemplo anterior:

```
String[] campos = new String[] {"usuario", "email"};
String[] args = new String[] {"usu1"};

Cursor c = db.query("Usuarios", campos, "usuario=?", args, null, null, null);

//Nos aseguramos de que existe al menos un registro
if (c.moveToFirst()) {
     //Recorremos el cursor hasta que no haya más registros
     do {
          String usuario = c.getString(0);
          String email = c.getString(1);
     } while(c.moveToNext());
}
```

Además de los métodos comentados de la clase Cursor existen muchos más que nos pueden ser útiles en muchas ocasiones. Por ejemplo, getCount() te dirá el número total de registros devueltos en el cursor, getColumnName(i) devuelve el nombre de la columna con índice i, moveToPosition(i) mueve el puntero del cursor al registro con índice i, etc. Podéis consultar la lista completa de métodos disponibles en la clase Cursor en la documentación oficial de Android.

Con esto, terminamos la serie de apartados básicos dedicados a las tareas de mantenimiento de datos en aplicaciones Android mediante bases de datos SQLite. Soy consciente de que dejamos en el tintero algunos temas algo más avanzados (como por ejemplo el uso de *transacciones*, que intentaré tratar más adelante), pero con los métodos descritos podremos realizar un porcentaje bastante alto de todas las tareas necesarias relativas al tratamiento de datos estructurados en aplicaciones Android.



Preferencias

VII. Preferencias en Android

Preferencias Compartidas

En el apartado anterior del Curso de Programación en Android vimos cómo construir un widget básico y prometimos dedicar un segundo apartado a comentar otras funcionalidades más avanzadas de este tipo de componentes. Sin embargo, antes de esto he decidido hacer un pequeño alto en el camino para hablar de un tema que nos será de ayuda más adelante, y no sólo para la construcción de widgets, sino para cualquier tipo de aplicación Android. Este tema es la administración de preferencias.

Las preferencias no son más que datos que una aplicación debe guardar para personalizar la experiencia del usuario, por ejemplo información personal, opciones de presentación, etc. En apartados anteriores vimos ya uno de los métodos disponibles en la plataforma Android para almacenar datos, como son las bases de datos SQLite. Las preferencias de una aplicación se podrían almacenar por su puesto utilizando este método, y no tendría nada de malo, pero Android proporciona otro método alternativo diseñado específicamente para administrar este tipo de datos: las preferencias compartidas o *shared preferences*. Cada preferencia se almacenará en forma de clave-valor, es decir, cada una de ellas estará compuesta por un identificador único (p.e. "email") y un valor asociado a dicho identificador (p.e. "prueba@email.com"). Además, y a diferencia de SQLite, los datos no se guardan en un fichero de binario de base de datos, sino en ficheros XML como veremos al final de este apartado.

La API para el manejo de estas preferencias es muy sencilla. Toda la gestión se centraliza en la clase `SharedPrefences`, que representará a una colección de preferencias. Una aplicación Android puede gestionar varias colecciones de preferencias, que se diferenciarán mediante un identificador único. Para obtener una referencia a una colección determinada utilizaremos el método `getSharedPrefences()` al que pasaremos el identificador de la colección y un modo de acceso. El modo de acceso indicará qué aplicaciones tendrán acceso a la colección de preferencias y qué operaciones tendrán permitido realizar sobre ellas. Así, tendremos tres posibilidades principales:

- `MODE_PRIVATE`. Sólo nuestra aplicación tiene acceso a estas preferencias.
- `MODE_WORLD_READABLE`. Todas las aplicaciones pueden leer estas preferencias, pero sólo la nuestra modificarlas.
- `MODE_WORLD_WRITABLE`. Todas las aplicaciones pueden leer y modificar estas preferencias.

Teniendo todo esto en cuenta, para obtener una referencia a una colección de preferencias llamada por ejemplo "*MisPreferencias*" y como modo de acceso exclusivo para nuestra aplicación haríamos lo siguiente:

```
SharedPreferences prefs =
     getSharedPreferences("MisPreferencias",Context.MODE_PRIVATE);
```

Una vez hemos obtenido una referencia a nuestra colección de preferencias, ya podemos obtener, insertar o modificar preferencias utilizando los métodos `get` o `put` correspondientes al tipo de dato de cada preferencia. Así, por ejemplo, para obtener el valor de una preferencia llamada "email" de tipo `String` escribiríamos lo siguiente:

```
SharedPreferences prefs =
     getSharedPreferences("MisPreferencias",Context.MODE_PRIVATE);

String correo = prefs.getString("email", "por_defecto@email.com");
```

Como vemos, al método `getString()` le pasamos el nombre de la preferencia que queremos recuperar y un segundo parámetro con un valor por defecto. Este valor por defecto será el devuelto por el método `getString()` si la preferencia solicitada no existe en la colección. Además del método `getString()`, existen por supuesto métodos análogos para el resto de tipos de datos básicos, por ejemplo `getInt()`, getLong(), `getFloat()`, `getBoolean()`, …

Para actualizar o insertar nuevas preferencias el proceso será igual de sencillo, con la única diferencia de que la actualización o inserción no la haremos directamente sobre el objeto `SharedPreferences`, sino sobre su objeto de edición `SharedPreferences.Editor`. A este último objeto accedemos mediante el método `edit()` de la clase `SharedPreferences`. Una vez obtenida la referencia al editor, utilizaremos los métodos `put` correspondientes al tipo de datos de cada preferencia para actualizar/insertar su valor, por ejemplo `putString(clave, valor)`, para actualizar una preferencia de tipo `String`. De forma análoga a los métodos `get` que ya hemos visto, tendremos disponibles métodos put para todos los tipos de datos básicos: `putInt()`, `putFloat()`, `putBoolean()`, etc. Finalmente, una vez actualizados/insertados todos los datos necesarios llamaremos al método `commit()` para confirmar los cambios. Veamos un ejemplo sencillo:

```
SharedPreferences prefs =
     getSharedPreferences("MisPreferencias",Context.MODE_PRIVATE);

SharedPreferences.Editor editor = prefs.edit();
editor.putString("email", "modificado@email.com");
editor.putString("nombre", "Prueba");
editor.commit();
```

¿Dónde se almacenan estas preferencias compartidas? Como dijimos al comienzo del apartado, las preferencias no se almacenan en ficheros binarios como las bases de datos SQLite, sino en ficheros XML. Estos ficheros XML se almacenan en una ruta con el siguiente patrón:

`/data/data/paquetejava/shared_prefs/nombre_coleccion.xml`

Así, por ejemplo, en nuestro caso encontraríamos nuestro fichero de preferencias en la ruta:

`/data/data/net.sgoliver.android/shared_prefs/MisPreferencias.xml`

Si descargamos este fichero desde el DDMS y lo abrimos con cualquier editor de texto veremos un contenido como el siguiente:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="nombre">prueba</string>
    <string name="email">modificado@email.com</string>
</map>
```

En este XML podemos observar cómo se han almacenado las dos preferencias de ejemplo que insertamos anteriormente, con sus claves y valores correspondientes.

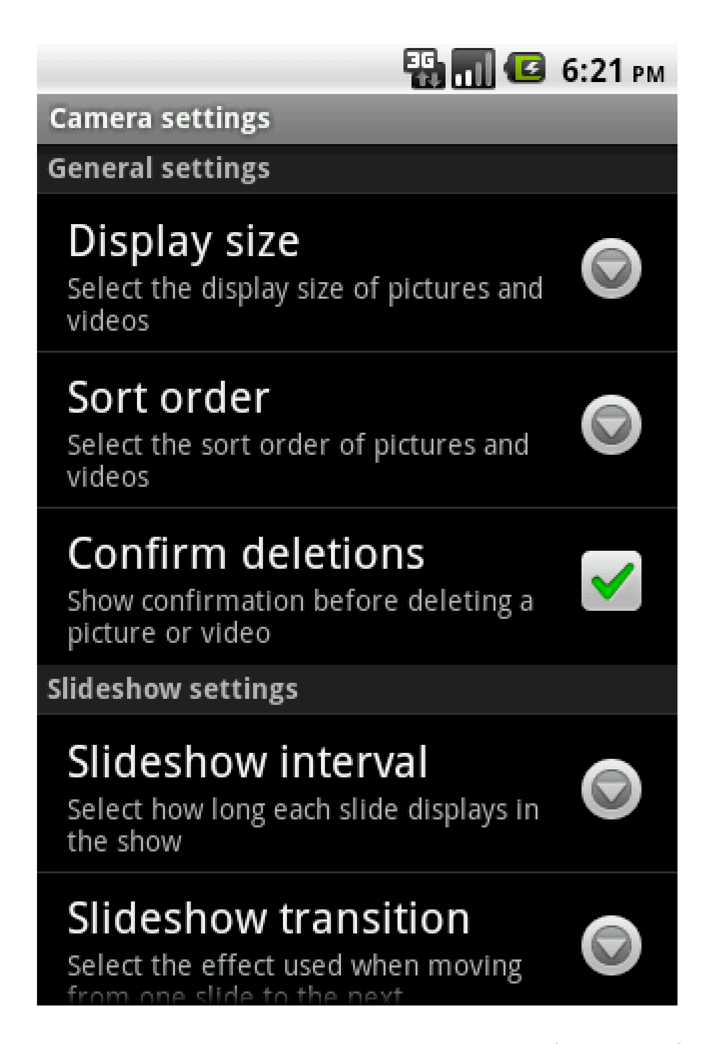
Y nada más, así de fácil y práctico. Con esto hemos aprendido una forma sencilla de almacenar determinadas opciones de nuestra aplicación sin tener que recurrir para ello a definir bases de datos SQLite, que aunque tampoco añaden mucha dificultad sí que requieren algo más de trabajo por nuestra parte.

En una segunda parte de este tema dedicado a las preferencias veremos cómo Android nos ofrece otra forma de gestionar estos datos, que se integra además fácilmente con la interfaz gráfica necesaria para solicitar los datos al usuario.

Pantallas de Preferencias

Ya hemos visto durante el curso algún apartado dedicado a las preferencias compartidas (*shared preferences*), un mecanismo que nos permite gestionar fácilmente las opciones de una aplicación permitiéndonos guardarlas en XML de una forma transparente para el programador. En aquel momento sólo vimos cómo hacer uso de ellas mediante código, es decir, creando nosotros mismos los objetos necesarios (`SharedPreferences`) y añadiendo, modificando y/o recuperando "a mano" los valores de las opciones a través de los métodos correspondientes (`getString()`, `putString()`, …). Sin embargo, ya avisamos de que Android ofrece una forma alternativa de definir mediante XML un conjunto de opciones para una aplicación y crear por nosotros las pantallas necesarias para permitir al usuario modificarlas a su antojo. A esto dedicaremos este segundo apartado sobre preferencias.

Si nos fijamos en cualquier pantalla de preferencias estándar de Android veremos que todas comparten una interfaz común, similar por ejemplo a la que se muestra en la imagen siguiente (correspondiente a la pantalla de opciones de la galería de imágenes de Android).



Si observamos la imagen anterior vemos cómo las diferentes opciones se organizan dentro de la **pantalla de opciones** en varias **categorías** ("*General Settings*" y "*Slideshow Settings*"). Dentro de cada categoría pueden aparecer varias opciones de diversos **tipos**, como por ejemplo de tipo checkbox ("*Confirm deletions*") o de tipo lista de selección ("*Display size*"). He resaltado las palabras "*pantalla de opciones*", "*categorías*", y "tipos de opción" porque serán estos los tres elementos principales con los que vamos a definir el conjunto de opciones o preferencias de nuestra aplicación. Empecemos.

Como hemos indicado, nuestra pantalla de opciones la vamos a definir mediante un XML, de forma similar a como definimos cualquier layout, aunque en este caso deberemos colocarlo en la carpeta /res/xml. El contenedor principal de nuestra pantalla de preferencias será el elemento <PreferenceScreen>. Este elemento representará a la pantalla de opciones en sí, dentro de la cual incluiremos el resto de elementos. Dentro de éste podremos incluir nuestra lista de opciones organizadas por categorías, que se representará mediante el elemento <PreferenceCategory> al que daremos un texto descriptivo utilizando su atributo android:title. Dentro de cada categoría podremos añadir cualquier número de opciones, las cuales pueden ser de distintos tipos, entre los que destacan:

Tipo	Descripción
CheckBoxPreference	Marca seleccionable.
EditTextPreference	Cadena simple de texto.
ListPreference	Lista de valores seleccionables (exclusiva).
MultiSelectListPreference	Lista de valores seleccionables (múltiple).

Cada uno de estos tipos de preferencia requiere la definición de diferentes atributos, que iremos viendo en los siguientes apartados.

CheckBoxPreference

Representa un tipo de opción que sólo puede tomar dos valores distintos: activada o desactivada. Es el equivalente a un control de tipo checkbox. En este caso tan sólo tendremos que especificar los atributos: nombre interno de la opción (`android:key`), texto a mostrar (`android:title`) y descripción de la opción (`android:summary`). Veamos un ejemplo:

```
<CheckBoxPreference
    android:key="opcion1"
    android:title="Preferencia 1"
    android:summary="Descripción de la preferencia 1" />
```

EditTextPreference

Representa un tipo de opción que puede contener como valor una cadena de texto. Al pulsar sobre una opción de este tipo se mostrará un cuadro de diálogo sencillo que solicitará al usuario el texto a almacenar. Para este tipo, además de los tres atributos comunes a todas las opciones (`key`, `title` y `summary`) también tendremos que indicar el texto a mostrar en el cuadro de diálogo, mediante el atributo `android:dialogTitle`. Un ejemplo sería el siguiente:

```
<EditTextPreference
   android:key="opcion2"
   android:title="Preferencia 2"
   android:summary="Descripción de la preferencia 2"
   android:dialogTitle="Introduce valor" />
```

ListPreference

Representa un tipo de opción que puede tomar como valor un elemento, y sólo uno, seleccionado por el usuario entre una lista de valores predefinida. Al pulsar sobre una opción de este tipo se mostrará la lista de valores posibles y el usuario podrá seleccionar uno de ellos. Y en este caso seguimos añadiendo atributos. Además de los cuatro ya comentados (`key`, `title`, `summary` y `dialogTitle`) tendremos que añadir dos más, uno de ellos indicando la lista de valores a visualizar en la lista y el otro indicando los valores internos que utilizaremos para cada uno de los valores de la lista anterior (Ejemplo: al usuario podemos mostrar una lista con los valores "*Español*" y "*Francés*", pero internamente almacenarlos como "*ESP*" y "*FRA*").

Estas listas de valores las definiremos también como ficheros XML dentro de la carpeta `/res/xml`. Definiremos para ello los recursos de tipos `<string-array>` necesarios, en este caso dos, uno para la lista de valores visibles y otro para la lista de valores internos, cada uno de ellos con su ID único correspondiente. Veamos cómo quedarían dos listas de ejemplo, en un fichero llamado "`codigospaises.xml`":

```
<?xml version="1.0" encoding="utf-8" ?>
<resources>
   <string-array name="pais">
      <item>España</item>
      <item>Francia</item>
      <item>Alemania</item>
   </string-array>
   <string-array name="codigopais">
      <item>ESP</item>
      <item>FRA</item>
      <item>ALE</item>
   </string-array>
</resources>
```

En la preferencia utilizaremos los atributos `android:entries` y `android:entryValues` para hacer referencia a estas listas, como vemos en el ejemplo siguiente:

```
<ListPreference
   android:key="opcion3"
   android:title="Preferencia 3"
   android:summary="Descripción de la preferencia 3"
   android:dialogTitle="Indicar Pais"
   android:entries="@array/pais"
   android:entryValues="@array/codigopais" />
```

MultiSelectListPreference

[A partir de Android 3.0.x / Honeycomb] Las opciones de este tipo son muy similares a las `ListPreference`, con la diferencia de que el usuario puede seleccionar varias de las opciones de la lista de posibles valores. Los atributos a asignar son por tanto los mismos que para el tipo anterior.

```
<MultiSelectListPreference
   android:key="opcion4"
   android:title="Preferencia 4"
   android:summary="Descripción de la preferencia 4"
   android:dialogTitle="Indicar Pais"
   android:entries="@array/pais"
   android:entryValues="@array/codigopais" />
```

Como ejemplo completo, veamos cómo quedaría definida una pantalla de opciones con las 3 primeras opciones comentadas (ya que probaré con Android 2.2), divididas en 2 categorías llamadas por simplicidad "*Categoría 1*" y "*Categoría 2*". Llamaremos al fichero "`opciones.xml`".

```
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory android:title="Categoría 1">
        <CheckBoxPreference
            android:key="opcion1"
            android:title="Preferencia 1"
            android:summary="Descripción de la preferencia 1" />
        <EditTextPreference
            android:key="opcion2"
            android:title="Preferencia 2"
            android:summary="Descripción de la preferencia 2"
            android:dialogTitle="Introduce valor" />
    </PreferenceCategory>
    <PreferenceCategory android:title="Categoría 2">
        <ListPreference
            android:key="opcion3"
            android:title="Preferencia 3"
            android:summary="Descripción de la preferencia 3"
            android:dialogTitle="Indicar Pais"
            android:entries="@array/pais"
            android:entryValues="@array/codigopais" />
    </PreferenceCategory>
</PreferenceScreen>
```

Ya tenemos definida la estructura de nuestra pantalla de opciones, pero aún nos queda un paso más para poder hacer uso de ella desde nuestra aplicación. Además de la definición XML de la lista de opciones, debemos implementar una nueva actividad, que será a la que hagamos referencia cuando queramos mostrar nuestra pantalla de opciones y la que se encargará internamente de gestionar todas las opciones, guardarlas, modificarlas, etc, a partir de nuestra definición XML.

Android nos facilita las cosas ofreciéndonos una clase de la que podemos derivar fácilmente la nuestra propia y que hace todo el trabajo por nosotros. Esta clase se llama `PreferenceActivity`. Tan sólo deberemos crear una nueva actividad (yo la he llamado `PantallaOpciones`) que extienda a esta clase e implementar su evento `onCreate()` para añadir una llamada al método `addPreferencesFromResource()`, mediante el que indicaremos el fichero XML en el que hemos definido la pantalla de opciones. Lo vemos mejor directamente en el código:

```
public class PantallaOpciones extends PreferenceActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        addPreferencesFromResource(R.xml.opciones);
    }
}
```

Así de sencillo, nuestra nueva actividad, al extender a `PreferenceActivity`, se encargará por nosotros de crear la interfaz gráfica de nuestra lista de opciones según la hemos definido en el XML y se preocupará por nosotros de mostrar, modificar y guardar las opciones cuando sea necesario tras la acción del usuario.

Por supuesto, tendremos que añadir esta actividad al fichero `AndroidManifest.xml`, al igual que cualquier otra actividad que utilicemos en la aplicación.

```
<activity android:name=".PantallaOpciones"
    android:label="@string/app_name">
</activity>
```

Ya sólo nos queda añadir a nuestra aplicación algún mecanismo para mostrar la pantalla de preferencias. Esta opción suele estar en un menú, pero por simplificar el ejemplo vamos a añadir simplemente un botón (`btnPreferencias`) que llame a la ventana de preferencias.

Al pulsar este botón llamaremos a la ventana de preferencias mediante el método `startActivity()`, como ya hemos visto en alguna ocasión, al que pasaremos como parámetros el contexto de la aplicación (nos vale con nuestra actividad principal) y la clase de la ventana de preferencias (`PantallaOpciones.class`).

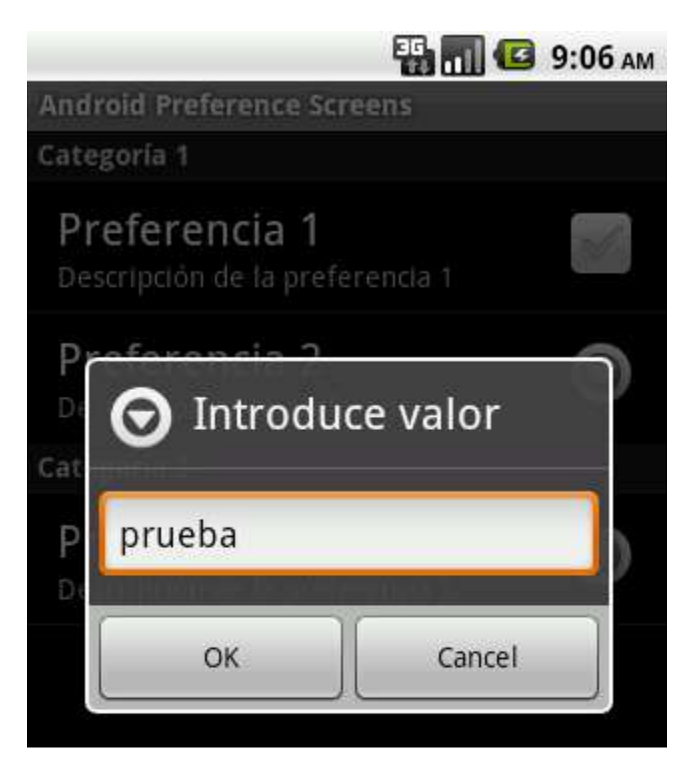
```
btnPreferencias = (Button)findViewById(R.id.BtnPreferencias);

btnPreferencias.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        startActivity(new Intent(AndroidPrefScreensActivity.this,
                                 PantallaOpciones.class));
    }
});
```

Y esto es todo, ya sólo nos queda ejecutar la aplicación en el emulador y pulsar el botón de preferencias para mostrar nuestra nueva pantalla de opciones. Debe quedar como muestra la imagen siguiente:



La primera opción podemos marcarla o desmarcarla directamente pulsando sobre la check de su derecha. La segunda, de tipo texto, nos mostrará al pulsarla un pequeño formulario para solicitar el valor de la opción.



Por último, la opción 3 de tipo lista, nos mostrará una ventana emergente con la lista de valores posibles, donde podremos seleccionar sólo uno de ellos.



Una vez establecidos los valores de las preferencias podemos salir de la ventana de opciones simplemente pulsando el botón Atrás del dispositivo o del emulador. Nuestra actividad `PantallaOpciones` se habrá ocupado por nosotros de guardar correctamente los valores de las opciones haciendo uso de la API de preferencias compartidas (*Shared Preferences*). Y para comprobarlo vamos a añadir otro botón (`btnObtenerOpciones`) a la aplicación de ejemplo que recupere el valor actual de las 3 preferencias y los escriba al log de la aplicación.

La forma de acceder a las preferencias compartidas de la aplicación ya la vimos en el apartado anterior sobre este tema. Obtenemos la lista de preferencias mediante el método `getDefaultSharedPreferences()` y posteriormente utilizamos los distintos métodos `get()` para recuperar el valor de cada opción dependiendo de su tipo.

```
btnObtenerPreferencias.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        SharedPreferences pref =
            PreferenceManager.getDefaultSharedPreferences(
                AndroidPrefScreensActivity.this);

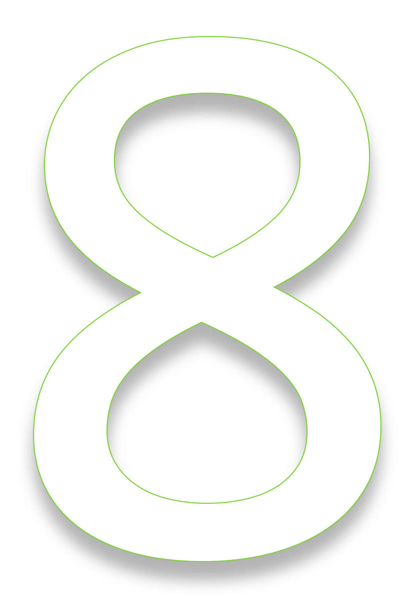
        Log.i("", "Opción 1: " + pref.getBoolean("opcion1", false));
        Log.i("", "Opción 2: " + pref.getString("opcion2", ""));
        Log.i("", "Opción 3: " + pref.getString("opcion3", ""));
    }
});
```

Si ejecutamos ahora la aplicación, establecemos las preferencias y pulsamos el nuevo botón de consulta que hemos creado veremos cómo en el log de la aplicación aparecen los valores correctos de cada preferencia. Se mostraría algo como lo siguiente:

```
10-08 09:27:09.681: INFO/(1162): Opción 1: true
10-08 09:27:09.681: INFO/(1162): Opción 2: prueba
10-08 09:27:09.693: INFO/(1162): Opción 3: FRA
```

Y hasta aquí hemos llegado con el tema de las preferencias, un tema muy interesante de controlar ya que casi ninguna aplicación se libra de hacer uso de ellas.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-pref-activity.zip



Localización Geográfica

VIII. Localización Geográfica

Localización Geográfica Básica

La localización geográfica en Android es uno de esos servicios que, a pesar de requerir poco código para ponerlos en marcha, no son para nada intuitivos ni fáciles de llegar a comprender por completo. Y esto no es debido al diseño de la plataforma Android en sí, sino a la propia naturaleza de este tipo de servicios. Por un lado, existen multitud de formas de obtener la localización de un dispositivo móvil, aunque la más conocida y popular es la localización por GPS, también es posible obtener la posición de un dispositivo por ejemplo a través de las antenas de telefonía móvil o mediante puntos de acceso Wi-Fi cercanos, y todos cada uno de estos mecanismos tiene una precisión, velocidad y consumo de recursos distinto. Por otro lado, el modo de funcionamiento de cada uno de estos mecanismos hace que su utilización desde nuestro código no sea todo lo directa e intuitiva que se desearía. Iremos comentando todo esto a lo largo del apartado, pero vayamos paso a paso.

¿Qué mecanismos de localización tenemos disponibles?

Lo primero que debe conocer una aplicación que necesite obtener la localización geográfica es qué mecanismos de localización (proveedores de localización, o *location providers*) tiene disponibles en el dispositivo. Como ya hemos comentado, los más comunes serán el GPS y la localización mediante la red de telefonía, pero podrían existir otros según el tipo de dispositivo.

La forma más sencilla de saber los proveedores disponibles en el dispositivo es mediante una llamada al método `getAllProviders()` de la clase `LocationManager`, clase principal en la que nos basaremos siempre a la hora de utilizar la API de localización de Android. Para ello, obtendremos una referencia al *location manager* llamando a `getSystemService(LOCATION_SERVICE)`, y posteriormente obtendremos la lista de proveedores mediante el método citado para obtener la lista de nombres de los proveedores:

```
LocationManager locManager =
    (LocationManager)getSystemService(LOCATION_SERVICE);
List<String> listaProviders = locManager.getAllProviders();
```

Una vez obtenida la lista completa de proveedores disponibles podríamos acceder a las propiedades de cualquiera de ellos (precisión, coste, consumo de recursos, o si es capaz de obtener la altitud, la velocidad, …). Así, podemos obtener una referencia al *provider* mediante su nombre llamando al método `getProvider(nombre)` y posteriormente utilizar los métodos disponibles para conocer sus propiedades, por ejemplo `getAccuracy()` para saber su precisión (tenemos disponibles las constantes `Criteria.ACCURACY_FINE` para precisión alta, y `Criteria.ACCURACY_COARSE` para precisión media), `supportsAltitude()` para saber si obtiene la altitud, o `getPowerRequirement()` para obtener el nivel de consumo de recursos del proveedor. La lista completa de métodos

para obtener las características de un proveedor se puede consultar en la documentación oficial de la clase `LocationProvider`.

```
LocationManager locManager =
    (LocationManager)getSystemService(LOCATION_SERVICE);
List<String> listaProviders = locManager.getAllProviders();

LocationProvider provider = locManager.getProvider(listaProviders.get(0));
int precision = provider.getAccuracy();
boolean obtieneAltitud = provider.supportsAltitude();
int consumoRecursos = provider.getPowerRequirement();
```

Al margen de esto, hay que tener en cuenta que la lista de proveedores devuelta por el método `getAllProviders()` contendrá todos los proveedores de localización conocidos por el dispositivo, incluso si éstos no están permitidos (según los permisos de la aplicación) o no están activados, por lo que esta información puede que no nos sea de mucha ayuda.

¿Qué proveedor de localización es mejor para mi aplicación?

Android proporciona un mecanismo alternativo para obtener los proveedores que cumplen unos determinados requisitos entre todos los disponibles. Para ello nos permite definir un criterio de búsqueda, mediante un objeto de tipo `Criteria`, en el que podremos indicar las características mínimas del proveedor que necesitamos utilizar (podéis consultar la documentación oficial de la clase `Criteria` para saber todas las características que podemos definir). Así, por ejemplo, para buscar uno con precisión alta y que nos proporcione la altitud definiríamos el siguiente criterio de búsqueda:

```
Criteria req = new Criteria();
req.setAccuracy(Criteria.ACCURACY_FINE);
req.setAltitudeRequired(true);
```

Tras esto, podremos utilizar los métodos `getProviders()` o `getBestProvider()` para obtener la lista de proveedores que se ajustan mejor al criterio definido o el proveedor que mejor se ajusta a dicho criterio, respectivamente. Además, ambos métodos reciben un segundo parámetro que indica si queremos que sólo nos devuelvan proveedores que están activados actualmente. Veamos cómo se utilizarían estos métodos:

```
//Mejor proveedor por criterio
String mejorProviderCrit = locManager.getBestProvider(req, false);

//Lista de proveedores por criterio
List<String> listaProvidersCrit = locManager.getProviders(req, false);
```

Con esto, ya tenemos una forma de seleccionar en cada dispositivo aquel proveedor que mejor se ajusta a nuestras necesidades.

¿Está disponible y activado un proveedor determinado?

Aunque, como ya hemos visto, tenemos la posibilidad de buscar dinámicamente proveedores de localización según un determinado criterio de búsqueda, es bastante común que nuestra aplicación esté diseñada para utilizar uno en concreto, por ejemplo el GPS, y por tanto

necesitaremos algún mecanismo para saber si éste está activado o no en el dispositivo. Para esta tarea, la clase `LocationManager` nos proporciona otro método llamado `isProviderEnabled()` que nos permite hacer exactamente lo que necesitamos. Para ello, debemos pasarle el nombre del *provider* que queremos consultar. Para los más comunes tenemos varias constantes ya definidas:

- `LocationManager.NETWORK_PROVIDER`. Localización por la red de telefonía.
- `LocationManager.GPS_PROVIDER`. Localización por GPS.

De esta forma, si quisiéramos saber si el GPS está habilitado o no en el dispositivo (y actuar en consecuencia), haríamos algo parecido a lo siguiente:

```
//Si el GPS no está habilitado
if (!locManager.isProviderEnabled(LocationManager.GPS_PROVIDER)) {
     mostrarAvisoGpsDeshabilitado();
}
```

En el código anterior, verificamos si el GPS está activado y en caso negativo mostramos al usuario un mensaje de advertencia. Este mensaje podríamos mostrarlo sencillamente en forma de notificación de tipo *toast*, pero en el próximo apartado sobre localización veremos cómo podemos, además de informar de que el GPS está desactivado, invitar al usuario a activarlo dirigiéndolo automáticamente a la pantalla de configuración del dispositivo.

El GPS ya está activado, ¿y ahora qué?

Una vez que sabemos que nuestro proveedor de localización favorito está activado, ya estamos en disposición de intentar obtener nuestra localización actual. Y aquí es donde las cosas empiezan a ser menos intuitivas. Para empezar, en Android no existe ningún método del tipo "obtenerPosiciónActual()". Obtener la posición a través de un dispositivo de localización como por ejemplo el GPS no es una tarea inmediata, sino que puede requerir de un cierto tiempo de procesamiento y de espera, por lo que no tendría sentido proporcionar un método de ese tipo.

Si buscamos entre los métodos disponibles en la clase `LocationManager`, lo más parecido que encontramos es un método llamado `getLastKnownLocation(String provider)`, que como se puede suponer por su nombre, nos devuelve la última posición conocida del dispositivo devuelta por un provider determinado. Es importante entender esto: este método NO devuelve la posición actual, este método NO solicita una nueva posición al proveedor de localización, este método se limita a devolver la última posición que se obtuvo a través del proveedor que se le indique como parámetro. Y esta posición se pudo obtener hace pocos segundos, hace días, hace meses, o incluso nunca (si el dispositivo ha estado apagado, si nunca se ha activado el GPS, …). Por tanto, cuidado cuando se haga uso de la posición devuelta por el método `getLastKnownLocation()`.

Entonces, ¿de qué forma podemos obtener la posición real actualizada? Pues la forma correcta de proceder va a consistir en algo así como activar el proveedor de localización y suscribirnos a sus notificaciones de cambio de posición. O dicho de otra forma, vamos a suscribirnos al evento que se lanza cada vez que un proveedor recibe nuevos datos sobre la

localización actual. Y para ello, vamos a darle previamente unas indicaciones (que no ordenes, ya veremos esto en el próximo apartado) sobre cada cuanto tiempo o cada cuanta distacia recorrida necesitaríamos tener una actualización de la posición.

Todo esto lo vamos a realizar mediante una llamada al método `requestLocationUpdates()`, al que deberemos pasar 4 parámetros distintos:

- Nombre del proveedor de localización al que nos queremos suscribir.
- Tiempo mínimo entre actualizaciones, en milisegundos.
- Distancia mínima entre actualizaciones, en metros.
- Instancia de un objeto `LocationListener`, que tendremos que implementar previamente para definir las acciones a realizar al recibir cada nueva actualización de la posición.

Tanto el tiempo como la distancia entre actualizaciones pueden pasarse con valor 0, lo que indicaría que ese criterio no se tendrá en cuenta a la hora de decidir la frecuencia de actualizaciones. Si ambos valores van a cero, las actualizaciones de posición se recibirán tan pronto y tan frecuentemente como estén disponibles. Además, como ya hemos indicado, es importante comprender que tanto el tiempo como la distancia especificadas se entenderán como simples indicaciones o "pistas" para el proveedor, por lo que puede que no se cumplan de forma estricta. En el próximo apartado intentaremos ver esto con más detalle para entenderlo mejor. Por ahora nos basta con esta información.

En cuanto al *listener*, éste será del tipo `LocationListener` y contendrá una serie de métodos asociados a los distintos eventos que podemos recibir del proveedor:

- `onLocationChanged(location)`. Lanzado cada vez que se recibe una actualización de la posición.
- `onProviderDisabled(provider)`. Lanzado cuando el proveedor se deshabilita.
- `onProviderEnabled(provider)`. Lanzado cuando el proveedor se habilita.
- `onStatusChanged(provider, status, extras)`. Lanzado cada vez que el proveedor cambia su estado, que puede variar entre `OUT_OF_SERVICE`, `TEMPORARILY_UNAVAILABLE`, `AVAILABLE`.

Por nuestra parte, tendremos que implementar cada uno de estos métodos para responder a los eventos del proveedor, sobre todo al más interesante, `onLocationChanged()`, que se ejecutará cada vez que se recibe una nueva localización desde el proveedor. Veamos un ejemplo de cómo implementar un listener de este tipo:

```
LocationListener locListener = new LocationListener() {

    public void onLocationChanged(Location location) {
        mostrarPosicion(location);
    }

    public void onProviderDisabled(String provider){
        lblEstado.setText("Provider OFF");
    }
```

```
    public void onProviderEnabled(String provider){
        lblEstado.setText("Provider ON");
    }

    public void onStatusChanged(String provider, int status, Bundle extras){
        lblEstado.setText("Provider Status: " + status);
    }
};
```

Como podéis ver, en nuestro caso de ejemplo nos limitamos a mostrar al usuario la información recibida en el evento, bien sea un simple cambio de estado, o una nueva posición, en cuyo caso llamamos al método auxiliar `mostrarPosicion()` para refrescar todos los datos de la posición en la pantalla. Para este ejemplo hemos construido una interfaz muy sencilla, donde se muestran 3 datos de la posición (latitud, longitud y precisión) y un campo para mostrar el estado del proveedor. Además, se incluyen dos botones para comenzar y detener la recepción de nuevas actualizaciones de la posición. No incluyo aquí el código de la interfaz para no alargar más el apartado, pero puede consultarse en el código fuente suministrado al final del texto. El aspecto de nuestra ventana es el siguiente:



En el método `mostrarPosicion()` nos vamos a limitar a mostrar los distintos datos de la posición pasada como parámetro en los controles correspondientes de la interfaz, utilizando para ello los métodos proporcionados por la clase `Location`. En nuestro caso utilizaremos `getLatitude()`, `getAltitude()` y `getAccuracy()` para obtener la latitud, longitud y precisión respectivamente. Por supuesto, hay otros métodos disponibles en esta clase para obtener la altura, orientación, velocidad, etc… que se pueden consultar en la documentación oficial de la clase `Location`. Veamos el código:

```
private void mostrarPosicion(Location loc) {
    if(loc != null)
    {
        lblLatitud.setText("Latitud: " + String.valueOf(loc.getLatitude()));
        lblLongitud.setText("Longitud: " +
String.valueOf(loc.getLongitude()));
        lblPrecision.setText("Precision: " +
String.valueOf(loc.getAccuracy()));
    }
    else
    {
        lblLatitud.setText("Latitud: (sin_datos)");
        lblLongitud.setText("Longitud: (sin_datos)");
        lblPrecision.setText("Precision: (sin_datos)");
    }
}
```

¿Por qué comprobamos si la localización recibida es `null`? Como ya hemos dicho anteriormente, no tenemos mucho control sobre el momento ni la frecuencia con la que vamos a recibir las actualizaciones de posición desde un proveedor, por lo que tampoco estamos seguros de tenerlas disponibles desde un primer momento. Por este motivo, una técnica bastante común es utilizar la posición que devuelve el método `getLastKnownLocation()` como posición "provisional" de partida y a partir de ahí esperar a recibir la primera actualización a través del `LocationListener`. Y como también dijimos, la última posición conocida podría no existir en el dispositivo, de ahí que comprobemos si el valor recibido es `null`. Para entender mejor esto, a continuación tenéis la estructura completa del método que lanzamos al comenzar la recepción de actualizaciones de posición (al pulsar el botón "*Activar*" de la interfaz):

```
private void comenzarLocalizacion()
{
    //Obtenemos una referencia al LocationManager
    locManager =
        (LocationManager)getSystemService(Context.LOCATION_SERVICE);

    //Obtenemos la última posición conocida
    Location loc =
        locManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);

    //Mostramos la última posición conocida
    mostrarPosicion(loc);

    //Nos registramos para recibir actualizaciones de la posición
    locListener = new LocationListener() {
        public void onLocationChanged(Location location) {
            mostrarPosicion(location);
        }

        //Resto de métodos del listener
        //...
    };

    locManager.requestLocationUpdates(
        LocationManager.GPS_PROVIDER, 30000, 0, locListener);
}
```

Como se puede observar, al comenzar la recepción de posiciones, mostramos en primer lugar la última posición conocida, y posteriormente solicitamos al GPS actualizaciones de posición cada 30 segundos.

Por último, nos quedaría únicamente comentar cómo podemos detener la recepción de nuevas actualizaciones de posición. Algo que es tan sencillo como llamar al método `removeUpdates()` del *location manager*. De esta forma, la implementación del botón "*Desactivar*" sería tan sencilla como esto:

```
btnDesactivar.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        locManager.removeUpdates(locListener);
    }
});
```

Con esto habríamos concluido nuestra aplicación de ejemplo. Sin embargo, si descargáis el código completo del apartado y ejecutáis la aplicación en el emulador veréis que, a pesar funcionar todo correctamente, sólo recibiréis una lectura de la posición (incluso puede que ninguna). Esto es debido a que la ejecución y prueba de aplicaciones de este tipo en el emulador de Android, al no tratarse de un dispositivo real y no estar disponible un receptor GPS, requiere de una serie de pasos adicionales para simular cambios en la posición del dispositivo.

Todo esto, además de algunas aclaraciones que nos han quedado pendientes en esta primera entrega sobre localización, lo veremos en el próximo apartado. Por ahora os dejo el código fuente completo para que podáis hacer vuestras propias pruebas.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-localizacion-1.zip

Profundizando en la Localización Geográfica

En el apartado anterior del curso de programación en Android comentamos los pasos básicos necesarios para construir aplicaciones que accedan a la posición geográfica del dispositivo. Ya comentamos algunas particularidades de este servicio de la API de Android, pero dejamos en el tintero algunas aclaraciones más detalladas y un tema importante y fundamental, como es la depuración de este tipo de aplicaciones que manejan datos de localización. En este nuevo apartado intentaré abarcar todos estos temas.

Como base para este apartado voy a utilizar la misma aplicación de ejemplo que construimos en la anterior entrega, haciendo tan sólo unas pequeñas modificaciones:

- Reduciremos el tiempo entre actualizaciones de posición a la mitad, 15 segundos, para evitar tiempos de espera demasiado largos durante la ejecución de la aplicación.
- Generaremos algunos mensajes de log en puntos clave del código para poder estudiar con más detalle el comportamiento de la aplicación en tiempo de ejecución.

La generación de mensajes de log resulta ser una herramienta perfecta a la hora de depurar aplicaciones del tipo que estamos tratando, ya que en estos casos el código no facilita demasiado la depuración típica paso a paso que podemos realizar en otras aplicaciones.

En nuestro caso de ejemplo sólo vamos a generar mensajes de log cuando ocurran dos circunstancias:

- Cuando el proveedor de localización cambie de estado, evento `onStatusChanged()`, mostraremos el nuevo estado.
- Cuando se reciba una nueva actualización de la posición, evento `onLocationChanged()`, mostraremos las nuevas coordenadas recibidas.

Nuestro código quedaría por tanto tal como sigue:

```
private void actualizarPosicion()
{
    //Obtenemos una referencia al LocationManager
    locationManager =
        (LocationManager)getSystemService(Context.LOCATION_SERVICE);

    //Obtenemos la última posición conocida
    Location location =
        locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);

    //Mostramos la última posición conocida
    muestraPosicion(location);

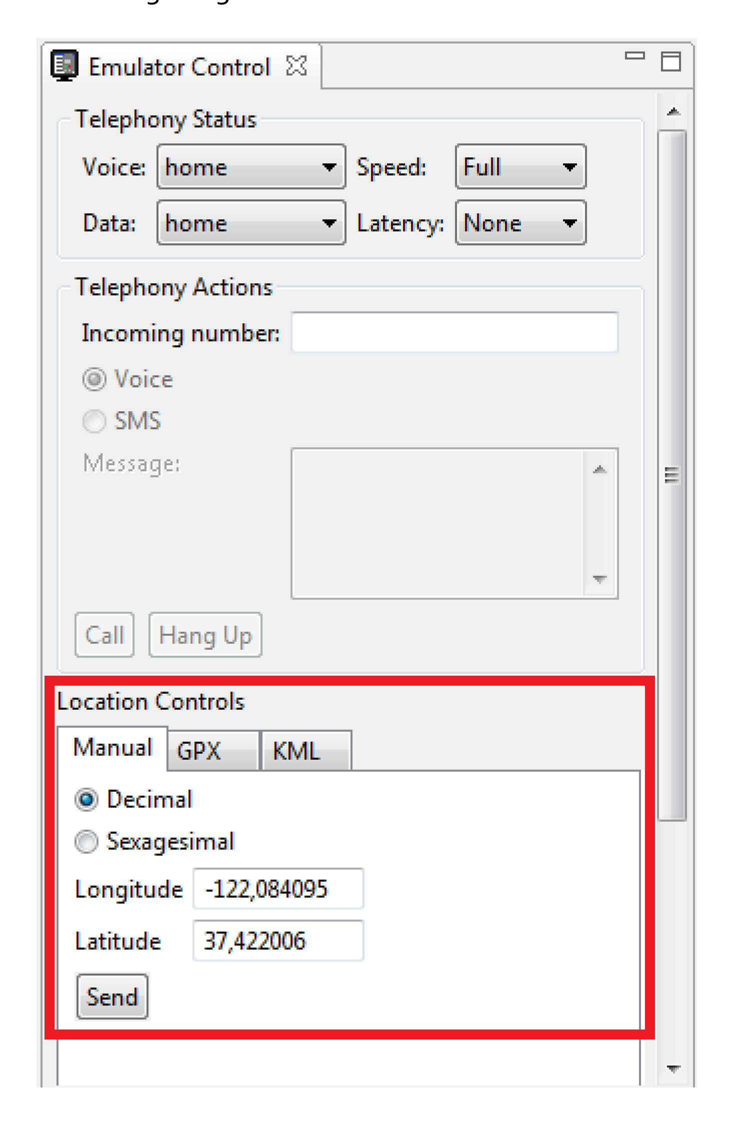
    //Nos registramos para recibir actualizaciones de la posición
    locationListener = new LocationListener() {
        public void onLocationChanged(Location location) {
            muestraPosicion(location);
        }
        public void onProviderDisabled(String provider){
            lblEstado.setText("Provider OFF");
        }
        public void onProviderEnabled(String provider){
            lblEstado.setText("Provider ON");
        }
        public void onStatusChanged(String provider, int status,
                                    Bundle extras){
            Log.i("LocAndroid", "Provider Status: " + status);
            lblEstado.setText("Provider Status: " + status);
        }
    };

    locationManager.requestLocationUpdates(
            LocationManager.GPS_PROVIDER, 15000, 0, locationListener);
}

private void muestraPosicion(Location loc) {
    if(loc != null) {
        lblLatitud.setText("Latitud: " + String.valueOf(loc.getLatitude()));
        lblLongitud.setText("Longitud: " +
                            String.valueOf(loc.getLongitude()));
        lblPrecision.setText("Precision: " +
                             String.valueOf(loc.getAccuracy()));
        Log.i("LocAndroid", String.valueOf(
            loc.getLatitude() + " - " + String.valueOf(loc.getLongitude())));
    } else {
        lblLatitud.setText("Latitud: (sin_datos)");
        lblLongitud.setText("Longitud: (sin_datos)");
        lblPrecision.setText("Precision: (sin_datos)");
    }
}
```

Si ejecutamos en este momento la aplicación en el emulador y pulsamos el botón "*Activar*" veremos cómo los cuadros de texto se rellenan con la información de la última posición conocida (si existe), pero sin embargo estos datos no cambiarán en ningún momento ya que por el momento el emulador de Android tan sólo cuenta con esa información. ¿Cómo podemos simular la actualización de la posición del dispositivo para ver si nuestra aplicación responde exactamente como esperamos?

Pues bien, para hacer esto tenemos varias opciones. La primera de ellas, y la más sencilla, es el envío manual de una nueva posición al emulador de Android, para simular que éste hubiera cambiado su localización. Esto se puede realizar desde la perspectiva de *DDMS*, en la pestaña *Emulator Control*, donde podemos encontrar una sección llamada *Location Controls*, mostrada en la imagen siguiente:



Con estos controles podemos enviar de forma manual al emulador en ejecución unas nuevas coordenadas de posición, para simular que éstas se hubieran recibido a través del proveedor de localización utilizado. De esta forma, si introducimos unas coordenadas de longitud y latitud y pulsamos el botón "*Send*" mientras nuestra aplicación se ejecuta en el emulador, esto provocará la ejecución del evento `onLocationChanged()` y por consiguiente se

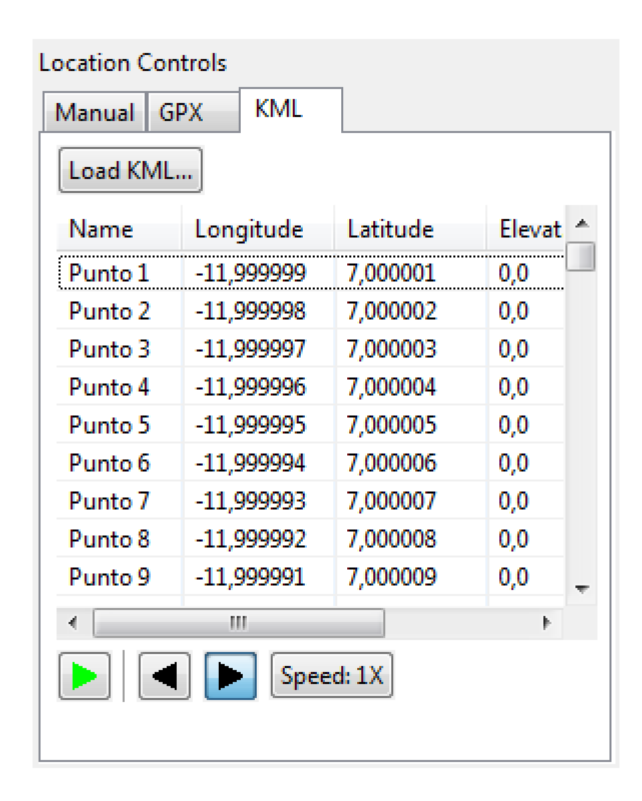
mostrarán estos mismos datos en sus controles correspondientes de la interfaz, como vemos en la siguiente captura de pantalla:



Por supuesto, si hacemos nuevos envíos de coordenadas desde Eclipse veremos cómo ésta se va actualizando en nuestra aplicación sin ningún tipo de problemas. Sin embargo este método de manual no resulta demasiado adecuado ni cómodo para probar toda la funcionalidad de nuestra aplicación, por ejemplo la actualización de posición cada 15 segundos.

Por ello, Android proporciona otro método algo menos manual de simular cambios frecuentes de posición para probar nuestras aplicaciones. Este método consiste en proporcionar, en vez de una sola coordenada cada vez, una lista de coordenadas que se irán enviando automáticamente al emulador una tras otra a una determinada velocidad, de forma que podamos simular que el dispositivo se mueve constantemente y que nuestra aplicación responde de forma correcta y en el momento adecuado a esos cambios de posición. Y esta lista de coordenadas se puede proporcionar de dos formas distintas, en formato GPX o como fichero KML. Ambos tipos de fichero son ampliamente utilizados por aplicaciones y dispositivos de localización, como GPS, aplicaciones de cartografía y mapas, etc. Los ficheros KML podemos generarlos por ejemplo a través de la aplicación *Google Earth* o manualmente con cualquier editor de texto, pero recomiendo consultar los dos enlaces anteriores para obtener más información sobre cada formato. Para este ejemplo, yo he generado un fichero KML de muestra con una lista de 1000 posiciones geográficas al azar.

Para utilizar este fichero como fuente de datos para simular cambios en la posición del dispositivo, accedemos nuevamente a los *Location Controls* y pulsamos sobre la pestaña GPX o KML, según el formato que hayamos elegido, que en nuestro caso será KML. Pulsamos el botón "*Load KML…*" para seleccionar nuestro fichero y veremos la lista de coordenadas como en la siguiente imagen:



Una vez cargado el fichero, tendremos disponibles los cuatro botones inferiores para (de izquierda a derecha):

- Avanzar automáticamente por la lista.
- Ir a la posición anterior de la lista de forma manual.
- Ir a la posición siguiente de la lista de forma manual.
- Establecer la velocidad de avance automático.

Entendido esto, vamos a utilizar la lista de posiciones para probar nuestra aplicación. Para ello, ejecutamos la aplicación en el emulador, pulsamos nuestro botón "*Activar*" para comenzar a detectar cambios de posición, y pulsamos el botón de avance automático (botón verde) que acabamos de comentar. Si observamos rápidamente la pantalla de nuestra aplicación veremos cómo se actualizan varias veces los valores de latitud y longitud actuales. ¿Cómo es posible? ¿No habíamos configurado el `LocationListener` para obtener actualizaciones de posición cada 15 segundos? Antes de contestar a esto, dejemos que la aplicación se ejecute durante un minuto más. Tras unos 60 segundos de ejecución detenemos la captura de posiciones pulsando nuestro botón "*Desactivar*".

Ahora vayamos a la ventana de log del DDMS y veamos los mensajes de log ha generado nuestra aplicación para intentar saber qué ha ocurrido. En mi caso, los mensajes generados son los siguientes (en tu caso deben ser muy parecidos):

```
05-08 10:50:37.921: INFO/LocAndroid(251): 7.0 - -11.999998333333334
05-08 10:50:38.041: INFO/LocAndroid(251): Provider Status: 2
05-08 10:50:38.901: INFO/LocAndroid(251): 7.000001666666666 - -11.999996666666668
05-08 10:50:39.941: INFO/LocAndroid(251): 7.000001666666666 - -11.999996666666668
05-08 10:50:41.011: INFO/LocAndroid(251): 7.000003333333333 - -11.999995000000002
05-08 10:50:43.011: INFO/LocAndroid(251): 7.000005000000001 - -11.999993333333334
05-08 10:50:45.001: INFO/LocAndroid(251): 7.000006666666667 - -11.999991666666665
05-08 10:50:46.061: INFO/LocAndroid(251): 7.000008333333333 - -11.999989999999999
05-08 10:50:47.131: INFO/LocAndroid(251): 7.000008333333333 - -11.999989999999999
05-08 10:50:47.182: INFO/LocAndroid(251): Provider Status: 1
05-08 10:51:02.232: INFO/LocAndroid(251): 7.000023333333333 - -11.999975
05-08 10:51:02.812: INFO/LocAndroid(251): 7.000023333333333 - -11.999973333333333
05-08 10:51:02.872: INFO/LocAndroid(251): Provider Status: 2
05-08 10:51:03.872: INFO/LocAndroid(251): 7.000024999999999 - -11.999973333333333
05-08 10:51:04.912: INFO/LocAndroid(251): 7.000026666666668 - -11.999971666666665
05-08 10:51:05.922: INFO/LocAndroid(251): 7.000026666666668 - -11.999971666666665
05-08 10:51:06.982: INFO/LocAndroid(251): 7.000028333333334 - -11.99997
05-08 10:51:08.032: INFO/LocAndroid(251): 7.000028333333334 - -11.999968333333333
05-08 10:51:09.062: INFO/LocAndroid(251): 7.00003 - -11.999968333333333
05-08 10:51:10.132: INFO/LocAndroid(251): 7.000031666666667 - -11.999966666666667
05-08 10:51:12.242: INFO/LocAndroid(251): 7.000033333333333 - -11.999965000000001
05-08 10:51:13.292: INFO/LocAndroid(251): 7.000033333333333 - -11.999963333333335
05-08 10:51:13.342: INFO/LocAndroid(251): Provider Status: 1
05-08 10:51:28.372: INFO/LocAndroid(251): 7.000048333333333 - -11.999950000000002
05-08 10:51:28.982: INFO/LocAndroid(251): 7.000048333333333 - -11.999950000000002
05-08 10:51:29.032: INFO/LocAndroid(251): Provider Status: 2
05-08 10:51:30.002: INFO/LocAndroid(251): 7.000050000000001 - -11.999948333333334
05-08 10:51:31.002: INFO/LocAndroid(251): 7.000051666666667 - -11.999946666666665
05-08 10:51:33.111: INFO/LocAndroid(251): 7.000053333333333 - -11.999944999999999
05-08 10:51:34.151: INFO/LocAndroid(251): 7.000053333333333 - -11.999944999999999
05-08 10:51:35.201: INFO/LocAndroid(251): 7.000055 - -11.999943333333333
05-08 10:51:36.251: INFO/LocAndroid(251): 7.0000566666666675 - -11.999941666666667
05-08 10:51:37.311: INFO/LocAndroid(251): 7.0000566666666675 - -11.999941666666667
05-08 10:51:38.361: INFO/LocAndroid(251): 7.0000583333333335 - -11.99994
05-08 10:51:38.431: INFO/LocAndroid(251): Provider Status: 1
```

Estudiemos un poco este log. Si observamos las marcas de fecha hora vemos varias cosas:

- Un primer grupo de actualizaciones entre las 10:50:37 y las 10:50:47, con 8 lecturas.
- Un segundo grupo de actualizaciones entre las 10:51:02 y las 10:51:13, con 11 lecturas.
- Un tercer grupo de actualizaciones entre las 10:51:28 y las 10:51:38, con 10 lecturas.
- Entre cada grupo de lecturas transcurren aproximadamente 15 segundos.
- Los grupos están formados por un número variable de lecturas.

Por tanto ya podemos sacar algunas conclusiones. Indicar al *location listener* una frecuencia de 15 segundos entre actualizaciones no quiere decir que vayamos a tener una sola lectura cada 15 segundos, sino que al menos tendremos una nueva con dicha frecuencia. Sin embargo, como podemos comprobar en los *logs*, las lecturas se recibirán por grupos separados entre sí por el intervalo de tiempo indicado.

Más conclusiones, ahora sobre el estado del proveedor de localización. Si buscamos en el log los momentos donde cambia el estado del proveedor vemos dos cosas importantes:

- Después de recibir cada grupo de lecturas el proveedor pasa a estado 1 (`TEMPORARILY_UNAVAILABLE`).
- Tras empezar a recibir de nuevo lecturas el proveedor pasa a estado 2 (`AVAILABLE`).

Estos cambios en el estado de los proveedores de localización pueden ayudarnos a realizar diversas tareas. Un ejemplo típico es utilizar el cambio de estado a 1 (es decir, cuando se ha terminado de recibir un grupo de lecturas) para seleccionar la lectura más precisa del grupo

recibido, algo especialmente importante cuando se están utilizando varios proveedores de localización simultáneamente, cada uno con una precisión distinta.

A modo de resumen, en este apartado hemos visto cómo podemos utilizar las distintas herramientas que proporciona la plataforma Android y el entorno de desarrollo Eclipse para simular cambios de posición del dispositivo durante la ejecución de nuestras aplicaciones en el emulador. También hemos visto cómo la generación de mensajes de log puede ayudarnos a depurar este tipo de aplicaciones, y finalmente hemos utilizado esta herramienta de depuración para entender mejor el funcionamiento de los *location listener* y el comportamiento de los proveedores de localización.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-localizacion-2.zip



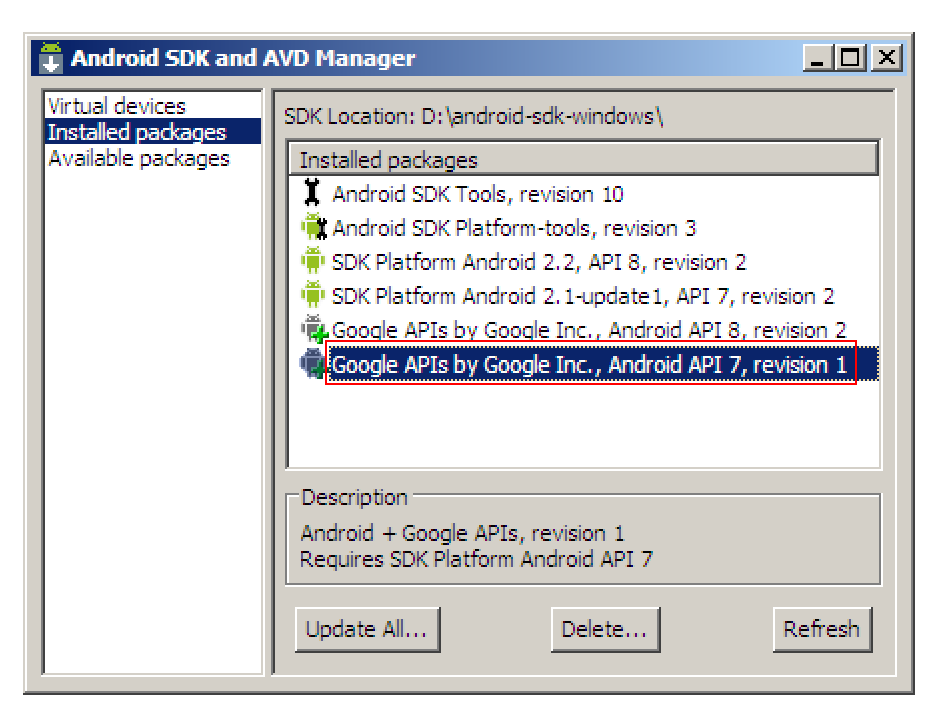
Mapas

IX. Mapas en Android

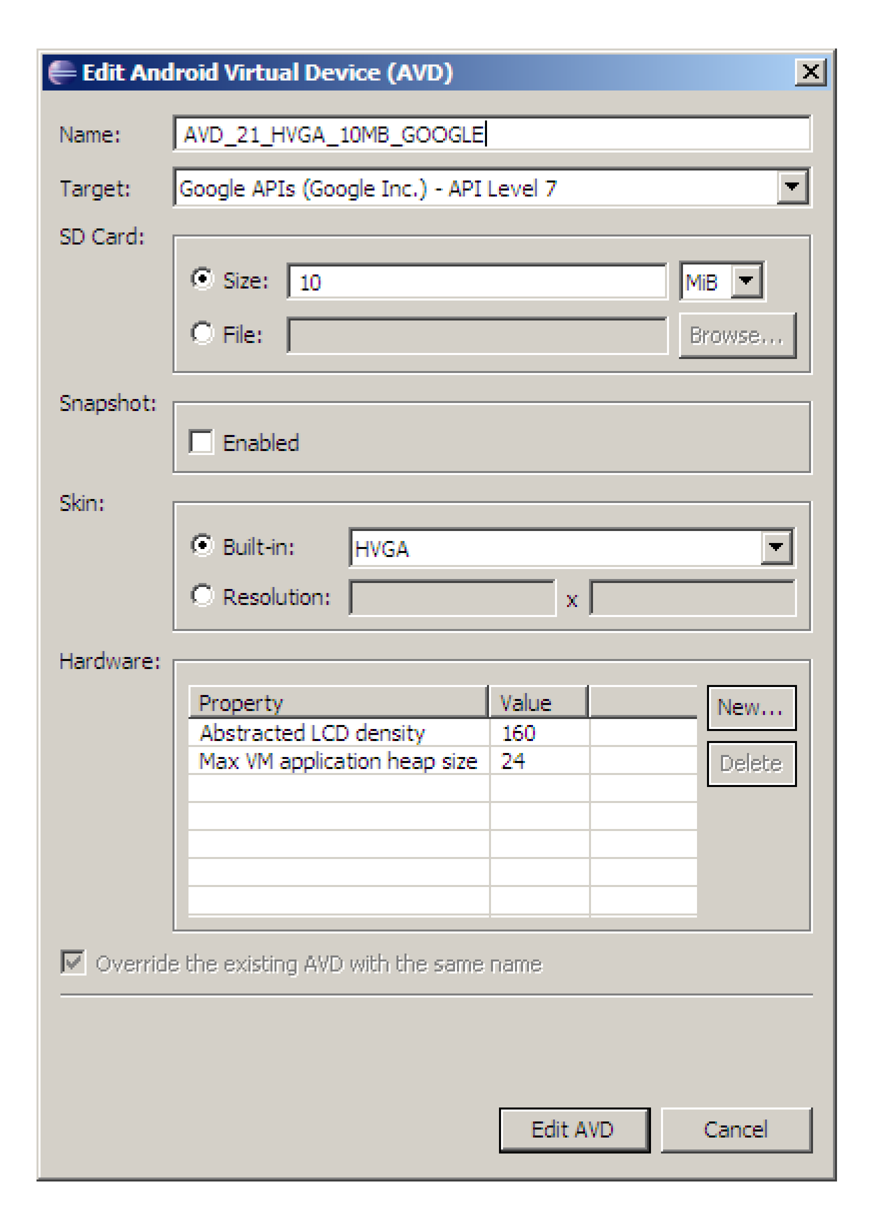
Preparativos y ejemplo básico

Siguiendo con nuestro Curso de Programación en Android, tras haber hablado en los dos últimos apartados sobre localización geográfica mediante GPS (aquí y aquí), en este nuevo apartado vamos a empezar a hablar de un complemento ideal para este tipo de servicios y aplicaciones, como es la utilización de mapas en nuestras aplicaciones haciendo uso de la API Android de *Google Maps*.

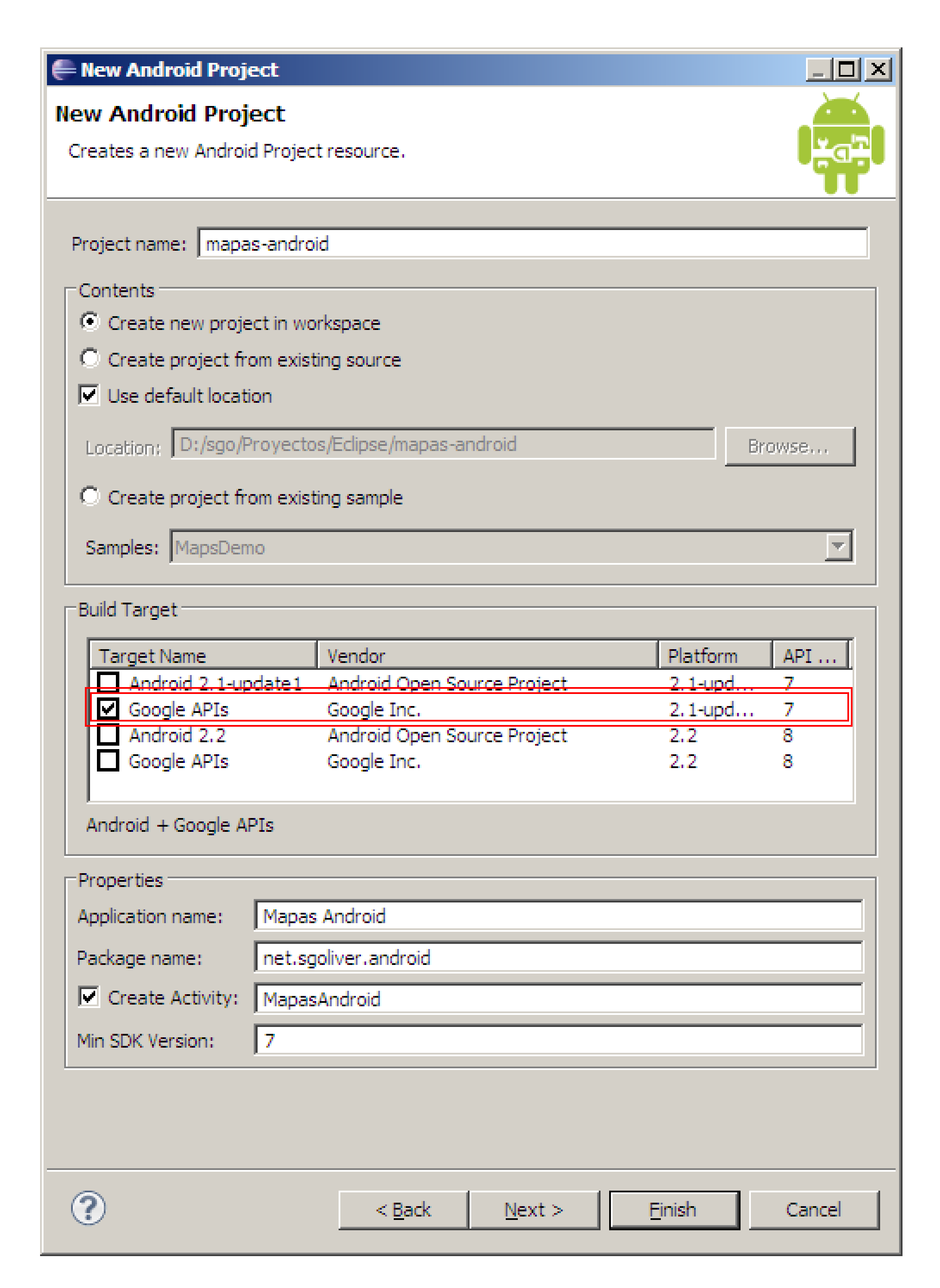
Antes de empezar a utilizar el servicio de mapas de Google es necesario realizar algunas tareas previas. En primer lugar, debemos asegurarnos de que tenemos instalado el paquete correspondiente a la versión de Android para la que desarrollamos "enriquecido" con las APIs de Google. Estos paquetes se llaman normalmente "*Google APIs by Google, Android API x, revisión y*". Esto podemos comprobarlo y descargarlo si es necesario desde Eclipse accediendo al *Android SDK and AVD Manager*. En mi caso, utilizaré el paquete correspondiente a Android 2.1 (API 7) + APIs de Google:



Para poder probar nuestras aplicaciones en el emulador también tendremos que crear un nuevo AVD que utilice este paquete como target:



Y por supuesto, al crear nuestro proyecto de Eclipse también tendremos que seleccionarlo como target en las propiedades:

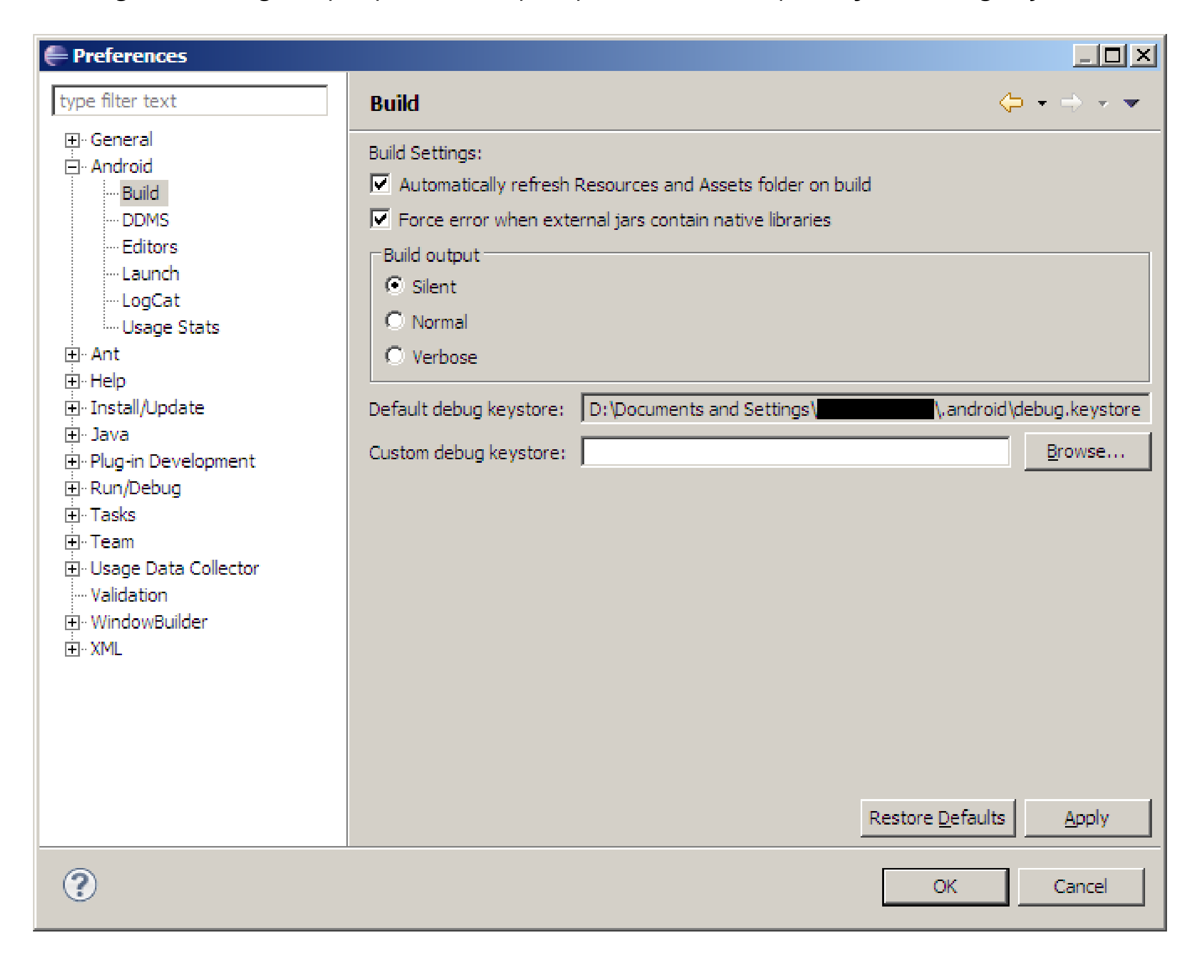


Con todo esto ya tendríamos creado nuestro proyecto de Eclipse y estaríamos preparados para poder ejecutar aplicaciones en el emulador sobre la versión correcta de Android y las APIs necesarias de Google. Por tanto, ya podemos centrarnos en la utilización de dichas APIs.

Esto último también requiere algunos pasos previos. Para poder utilizar la API de *Google Maps* se requiere la obtención previa de una clave de uso (API Key), que estará asociada al certificado con el que firmamos digitalmente nuestra aplicación. Esto quiere decir que si cambiamos el certificado con el que firmamos nuestra aplicación (algo que se hace normalmente como paso previo a la publicación de la aplicación en el Market) tendremos que cambiar también la clave de uso de la API.

En el tema de los certificados no voy a entrar mucho puesto que lo trataremos en un apartado posterior, por ahora tan sólo diremos que durante la construcción y depuración de nuestras aplicaciones en el emulador de Android se utiliza automáticamente un certificado de depuración creado por defecto. Por tanto, para poder depurar aplicaciones en el emulador que hagan uso de Google Maps tendremos que solicitar una clave asociada a nuestro certificado de depuración.

Para ello, en primer lugar debemos localizar el fichero donde se almacenan los datos de nuestro certificado de depuración, llamado *debug.keystore*. Podemos saber la ruta de este fichero accediendo a las preferencias de Eclipse, sección Android, apartado *Build* (mostrado en la siguiente imagen), y copiar la ruta que aparece en el campo "*Default Debug Keystore*":



Una vez conocemos la localización del fichero `debug.keystore`, vamos a acceder a él mediante la herramienta `keytool.exe` de java para obtener el hash MD5 del certificado. Esto lo haremos desde la línea de comandos de Windows mediante el siguiente comando:

```
C:\WINDOWS\system32\cmd.exe

C:\Program Files\Java\jre6\bin>keytool -list -alias androiddebugkey -keystore "D
:\Documents and Settings\[redacted]\.android\debug.keystore" -storepass andr
oid -keypass android
androiddebugkey, 13-dic-2010, PrivateKeyEntry,
Huella digital de certificado (MD5): A8:1E:57:5D:AF:1F:47:[redacted]

C:\Program Files\Java\jre6\bin>
```

Copiaremos el dato que aparece identificado como "*Huella digital de certificado (MD5)*" y con éste accederemos a la web de Google (http://code.google.com/android/maps-api-signup.html) para solicitar una clave de uso de la API de Google Maps para depurar nuestras aplicaciones (Insisto, si posteriormente vamos a publicar nuestra aplicación en el Market deberemos solicitar otra clave asociada al certificado real que utilicemos). Dicha web nos solicitará la marca MD5 de nuestro certificado y nos proporcionará la clave de uso de la API, como se muestra en la siguiente imagen:

API de Google Maps

Página principal de Google Code > API de Google Maps > Suscripción al API de Google Maps

Gracias por suscribirte a la clave del API de Android Maps.

Tu clave es:

```
0ss-5q6s3FKYkkp3atMUH[redacted]
```

Esta clave es válida para todas las aplicaciones firmadas con el certificado cuya huella dactilar sea:

```
A8:1E:57:5D:AF:1F:47:[redacted]
```

Incluimos un diseño xml de ejemplo para que puedas iniciarte por los senderos de la creación de mapas:

```
<com.google.android.maps.MapView
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:apiKey="0ss-5q6s3FKYkkp3atMUH[redacted]"
   />
```

Consulta la documentación del API para obtener más información.

Con esto, terminamos con todos los pasos previos para la utilización de los servicios de Google Maps dentro de nuestras aplicaciones Android.

Una vez contamos con la clave de uso de la API, la inclusión de mapas en nuestra aplicación es una tarea relativamente sencilla y directa. En este primer apartado sobre mapas nos vamos a limitar a mostrar el mapa en la pantalla inicial de la aplicación, habilitaremos su manipulación (desplazamientos y zoom), y veremos cómo centrarlo en unas coordenadas concretas. En próximos apartados aprenderemos a manipular de forma dinámica estos mapas, a mostrar marcas sobre ellos, y a realizar otras operaciones más avanzadas.

Para incluir un mapa de *Google Maps* en una ventana de nuestra aplicación utilizaremos el control `MapView`. Estos controles se pueden añadir al layout de la ventana como cualquier otro control visto hasta el momento, tan sólo teniendo en cuenta que tendremos que indicar la clave de uso de *Google Maps* en su propiedad `android:apiKey` como se muestra a continuación:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

<com.google.android.maps.MapView
    android:id="@+id/mapa"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:apiKey="0ss-5q6s3FKYkkp3atMUH..."
    android:clickable="true" />

</LinearLayout>
```

Como vemos, además de la propiedad `apiKey`, también hemos establecido la propiedad `clickable` con valor `true`, de forma que podamos interactuar con el control mediante gestos con el dedo (por ejemplo, para desplazarnos por el mapa).

Este tipo de controles tienen la particularidad de que sólo pueden ser añadidos a una actividad de tipo `MapActivity`, por lo que pantalla de la aplicación en la que queramos incluir el mapa debe heredar de esta clase. Así, para nuestro caso de ejemplo, vamos a hacer que la clase principal herede de `MapActivity`, como vemos en el siguiente código:

```
package net.sgoliver.android;

import com.google.android.maps.MapActivity;
import com.google.android.maps.MapView;
import android.os.Bundle;

public class AndroidMapas extends MapActivity {

    private MapView mapa = null;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        //Obtenemos una referencia al control MapView
        mapa = (MapView)findViewById(R.id.mapa);

        //Mostramos los controles de zoom sobre el mapa
```

```
        mapa.setBuiltInZoomControls(true);
    }

    @Override
    protected boolean isRouteDisplayed() {
        return false;
    }
}
```

Como vemos, si nuestra clase hereda de `MapActivity` debemos implementar obligatoriamente el método `isRouteDisplayed()`, cuyo valor de retorno debe ser true sólo en caso de que vayamos a representar algún tipo de información de ruta sobre el mapa (esto no se trata de ningún tema técnico, sino tan sólo legal, para cumplir los términos de uso de la API de *Google Maps*). En este primer apartado nos limitaremos a mostrar el mapa en la pantalla principal de la aplicación, por lo que por el momento devolveremos false.

Además de esto, en el método `onCreate()` llamaremos al método `setBuiltInZoomControls()` sobre la referencia al control `MapView` para mostrar los controles de zoom estándar sobre el mapa, de forma que podamos acercar y alejar la vista del mapa. Con esto, ya tendríamos un mapa completamente funcional funcionando en nuestra aplicación.

Para ejecutar la aplicación de ejemplo sobre el emulador de Android aún nos queda un paso más, modificar el fichero `AndroidManifest.xml`. Por un lado, tendremos que especificar que vamos a hacer uso de la API de *Google Maps* (mediante la cláusula `<uses-library>`), y en segundo lugar necesitaremos habilitar los permisos necesarios para acceder a Internet (mediante la cláusula `<uses-permission>`). En el siguiente fragmento de código vemos cómo quedaría nuestro `AndroidManifest` tras añadir estas dos líneas:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="net.sgoliver.android"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-sdk android:minSdkVersion="7" />

    <application
        android:icon="@drawable/icon"
        android:label="@string/app_name">

        <uses-library android:name="com.google.android.maps" />

        <activity android:name=".AndroidMapas"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

    <uses-permission android:name="android.permission.INTERNET" />

</manifest>
```

Tras estos dos cambios ya podemos ejecutar el proyecto de Eclipse sobre el emulador. Comprobaremos cómo podemos movernos por el mapa y hacer zoom con los controles correspondientes:



En los próximos apartados aprenderemos a manipular las características de este mapa desde nuestro código a través de sus métodos y propiedades, veremos cómo añadir marcas visuales para resaltar lugares específicos en el mapa, y comentaremos algunas otras opciones más avanzadas.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-mapas-1.zip

Control MapView

En el apartado anterior del curso, ya dimos los primeros pasos necesarios para configurar un proyecto de Eclipse de forma que pudiéramos utilizar la API de *Google Maps* desde nuestras aplicaciones. Vimos además cómo crear una aplicación de ejemplo muy básica en la que mostrábamos un mapa y permitíamos su manipulación mediante los gestos del usuario o los controles de zoom por defecto.

En este apartado nos centraremos en comentar los diferentes métodos y propiedades del control MapView con los que podremos manipular un mapa desde el código de nuestra aplicación.

Veamos en primer lugar cómo ajustar algunas propiedades del control para ajustarlo a nuestras necesidades. Por defecto, cuando añadimos un control MapView a nuestra aplicación éste muestra en el modo de mapa tradicional, sin embargo, el control también nos permite cambiar por ejemplo a vista de satélite, marcar las zonas disponibles en *StreetView*, o mostrar la información del tráfico. Esto lo conseguimos mediante los siguientes métodos:

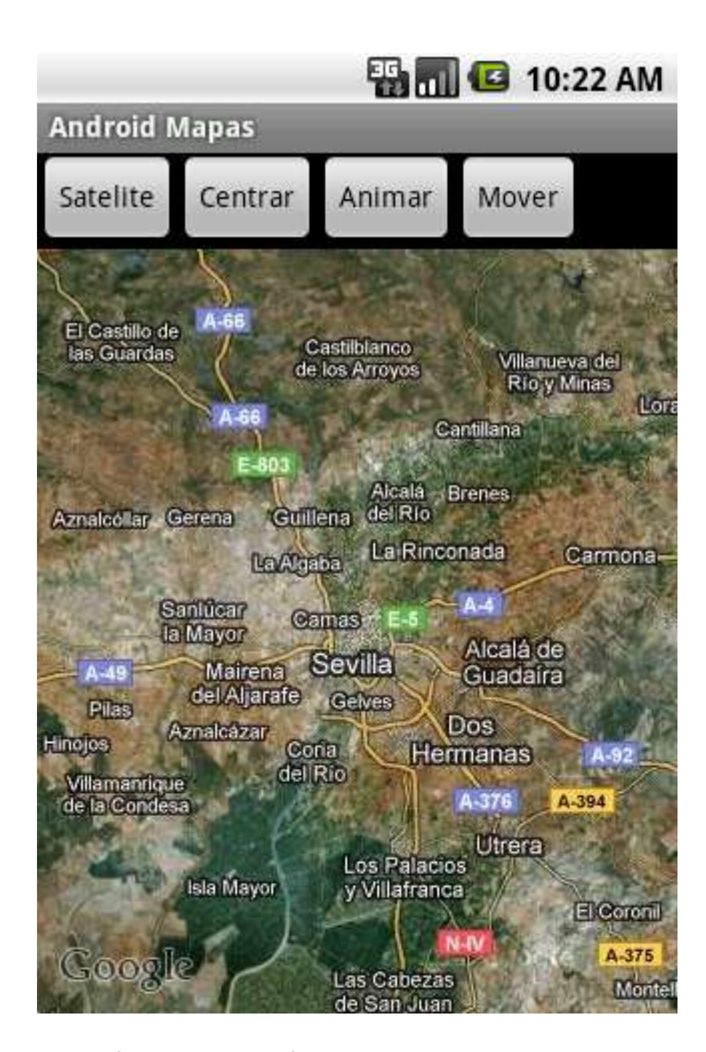
- setSatellite(true)
- setStreetView(true)
- setTraffic(true)

Por supuesto existen también otros tres métodos para consultar el estado de cada uno de estos modos: isSatellite(), isStreetView() e isTraffic().

En este apartado voy a continuar ampliando la aplicación de ejemplo que construimos en el apartado anterior, al que añadiremos varios botones para realizar diferentes acciones. Así, por ejemplo vamos a incluir un botón para alternar en nuestro mapa entre vista normal y vista de satélite.

```
private Button btnSatelite = null;
//...
btnSatelite = (Button)findViewById(R.id.BtnSatelite);
//...
btnSatelite.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        if(mapa.isSatellite())
            mapa.setSatellite(false);
        else
            mapa.setSatellite(true);
    }
});
```

Si ejecutamos la aplicación en este momento y probamos el nuevo botón veremos cómo se visualiza sin problemas nuestro mapa en vista de satélite:



Además de la elección del tipo de vista de mapa a mostrar en el control, también podemos elegir si queremos mostrar controles de zoom sobre el mapa. Para ello llamaremos al método `setBuiltInZoomControls()` indicando si queremos que se muestren o no los controles de zoom:

```
//Mostramos los controles de zoom sobre el mapa
mapa.setBuiltInZoomControls(true);
```

Al margen de las propiedades para personalizar el aspecto gráfico del control también dispondremos de varios métodos para consultar la información geográfica visualizada en el mismo. Así, por ejemplo, podremos saber las coordenadas geográficas en las que está centrado actualmente el mapa [método `getMapCenter()`] y el nivel de zoom que está aplicado [método `getZoomLevel()`].

```
//Obtenemos el centro del mapa
GeoPoint loc = mapa.getMapCenter();

//Latitud y Longitud (en micrograds)
int lat = loc.getLatitudeE6();
int lon = loc.getLongitudeE6();

//Nivel de zoom actual
int zoom = mapa.getZoomLevel();
```

Como vemos en el código anterior, las coordenadas del centro del mapa se obtienen en forma de objeto `GeoPoint`, que encapsula los valores de latitud y longitud expresados en microgrados (grados * 1E6). Estos valores se pueden obtener mediante los métodos `getLatitudeE6()` y `getLongitudeE6()` respectivamente.

Por su parte, el nivel de zoom se obtiene como un valor entero entre 1 y 21, siendo 21 el que ofrece mayor nivel de detalle en el mapa.

Sin embargo, si miramos la documentación de la API de la clase `MapView` veremos que no existen métodos para modificar estos valores de centro y zoom del mapa. ¿Cómo podemos entonces modificar la localización mostrada en el mapa? Pues bien, para esto habrá que acceder en primer lugar al controlador del mapa, algo que conseguimos mediante el método `getController()`. Este método nos devolverá un objeto `MapController` con el que sí podremos modificar los datos indicados. Así, contaremos con los métodos `setCenter()` y `setZoom()` al que podremos indicar las coordenadas centrales del mapa y el nivel de zoom deseado.

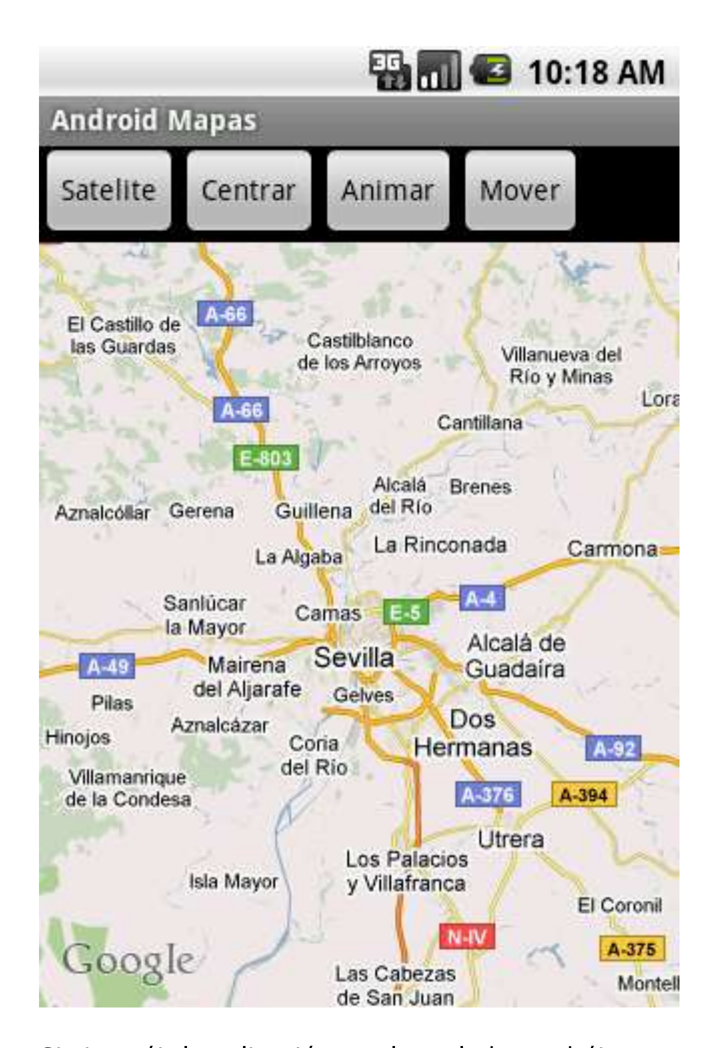
Vamos a incluir en la aplicación de ejemplo un nuevo botón para centrar el mapa sobre un punto determinado (Sevilla) y aplicaremos un nivel de zoom (10) que nos permita distinguir algo de detalle.

```
private Button btnCentrar = null;
private MapController controlMapa = null;
//...
btnCentrar = (Button)findViewById(R.id.BtnCentrar);
//...
controlMapa = mapa.getController();
//...
btnCentrar.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        Double latitud = 37.40*1E6;
        Double longitud = -5.99*1E6;

        GeoPoint loc =
            new GeoPoint(latitud.intValue(), longitud.intValue());

        controlMapa.setCenter(loc);
        controlMapa.setZoom(10);
    }
});
```

Como vemos, para establecer el punto central del mapa construiremos un objeto `GeoPoint` a partir de los valores de latitud y longitud y lo pasaremos como parámetro al método `setCenter()`.



Si ejecutáis la aplicación en el emulador podréis comprobar que funciona perfectamente de la forma esperada, sin embargo, el desplazamiento y zoom a la posición y nivel indicados se hace de forma instantánea, sin ningún tipo de animación.

Para resolver esto, la API nos ofrece otra serie de métodos que nos permitirán desplazarnos a una posición específica de forma progresiva y a subir o bajar el nivel de zoom de forma "animada", tal como se realiza al pulsar los botones de zoom del propio mapa. Estos métodos son `animateTo(GeoPoint)`, que desplazará el mapa hasta un punto determinado, y `zoomIn()`/`zoomOut()` que aumentará o disminuirá en 1 el nivel de zoom de forma progresiva.

Teniendo esto en cuenta, añadamos un nuevo botón para hacer algo análogo al anterior pero de forma progresiva:

```
private Button btnAnimar = null;
//...
btnAnimar = (Button)findViewById(R.id.BtnAnimar);
//...
btnAnimar.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        Double latitud = 37.40*1E6;
        Double longitud = -5.99*1E6;
```

```
        GeoPoint loc =
            new GeoPoint(latitud.intValue(), longitud.intValue());

        controlMapa.animateTo(loc);

        int zoomActual = mapa.getZoomLevel();

        for(int i=zoomActual; i<10; i++)
            controlMapa.zoomIn();
    }
});
```

Por último, disponemos de otro método que en ocasiones puede ser interesante y que nos permitirá desplazar el mapa, no a una posición específica, sino un determinado número de pixeles en una u otra dirección, al igual que conseguiríamos mediante gestos con el dedo sobre el mapa. Este método se llama `scrollBy()` y recibe como parámetros el número de pixeles que queremos desplazarnos en horizontal y en vertical.

Así, por ejemplo, podemos añadir un nuevo botón a la aplicación de ejemplo, que desplace el mapa 40 píxeles hacia la derecha y hacia abajo, de la siguiente forma:

```
private Button btnMover = null;
//...
btnMover = (Button)findViewById(R.id.BtnMover);
//...
btnMover.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        controlMapa.scrollBy(40, 40);
    }
});
```

En este apartado hemos aprendido a manipular desde nuestro código un control `MapView`, tanto a través de sus propiedades como de las acciones disponibles desde su controlador. En el próximo apartado veremos cómo podemos añadir capas a nuestro mapa de forma que podamos dibujar sobre él por ejemplo marcas de posición, o responder eventos de pulsación sobre el control.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-mapas-2.zip

Overlays (Capas)

Seguimos con nuestro Curso de Programación Android. En los dos apartados anteriores dedicados a la visualización y manipulación de mapas en nuestras aplicaciones ya vimos cómo mostrar de forma básica un mapa y cómo manipularlo desde nuestro código para realizar las acciones más frecuentes, como por ejemplo centrarlo o desplazarlo a una posición determinada o establecer el nivel de zoom. Nos quedó pendiente comentar cómo podemos añadir nuestra propia información a un mapa y cómo podemos responder a eventos de pulsación sobre el control. En este apartado nos ocuparemos de ambas cosas.

Empecemos por el principio. Para incluir información personalizada sobre un control MapView necesitaremos añadir sobre éste nuevas capas (*overlays*), donde dibujaremos toda la información adicional. Se puede incluir cualquier tipo de información en estas nuevas capas, por ejemplo indicaciones de ruta, marcadores, notas de texto…

El primer paso para definir una nueva capa de información será crear una clase java que derive de la clase Overlay. En esta nueva clase sobrescribiremos el método draw(), que es el lugar donde deberemos dibujar toda la información a incluir sobre el mapa en esta capa (por supuesto podemos añadir varias capas al mapa).

Por mostrar un ejemplo vamos a seguir trabajando con la misma aplicación de ejemplo del apartado anterior, al que añadiremos algún marcador sobre el mapa. Para no complicar mucho el ejemplo, añadiremos tan sólo un marcador sobre unas coordenadas fijas (las mismas coordenadas sobre las que aparece centrado el mapa cuando se inicia la aplicación).

El método draw() recibe como parámetro un objeto Canvas sobre el que podemos dibujar directamente utilizando los métodos de dibujo que ya hemos utilizado en otras ocasiones (drawLine(), drawCircle(), drawText(), drawBitmap()…). Sin embargo, a todos estos métodos de dibujo hay que indicarles las coordenadas en pixeles relativos a los bordes del control sobre el que se va a dibujar. Sin embargo, nosotros lo que tenemos en principio son unas coordenadas de latitud y longitud expresadas en grados. ¿Cómo podemos traducir entre unas unidades y otras para saber dónde dibujar nuestros marcadores? Pues bien, para solucionar esto Android nos proporciona la clase Projection, con la cual podremos hacer conversiones precisas entre ambos sistemas de referencia.

Veamos lo sencillo que es utilizar esta clase Projection. Partiremos de nuestros valores de latitud y longitud expresados en grados y a partir de ellos crearemos un objeto GeoPoint que los encapsule. Por otro lado, crearemos un nuevo objeto Projection mediante el método getProjection() de la clase MapView (objeto recibido también como parámetro del método draw()). Este nuevo objeto Projection creado tendrá en cuenta la posición sobre la que está centrada el mapa en este momento y el nivel de zoom aplicado para convertir convenientemente entre latitud-longitud en grados y coordenadas x-y en pixeles. Para hacer la conversión, llamaremos al método toPixels() que devolverá el resultado sobre un objeto Point de salida.

```
Double latitud = 37.40*1E6;
Double longitud = -5.99*1E6;

Projection projection = mapView.getProjection();
GeoPoint geoPoint =
    new GeoPoint(latitud.intValue(), longitud.intValue());

Point centro = new Point();
projection.toPixels(geoPoint, centro);
```

Una vez que tenemos las coordenadas convertidas a pixeles, ya podemos dibujar sobre ellas utilizando cualquier método de dibujo. Veamos en primer lugar cómo podríamos por ejemplo añadir un círculo y una etiqueta de texto sobre las coordenadas calculadas:

```
//Definimos el pincel de dibujo
Paint p = new Paint();
p.setColor(Color.BLUE);

//Marca Ejemplo 1: Círculo y Texto
canvas.drawCircle(centro.x, centro.y, 5, p);
canvas.drawText("Sevilla", centro.x+10, centro.y+5, p);
```

Para que nadie se pierda, veamos el código completo de nuestra clase derivada de `Overlay` (en un alarde de creatividad la he llamado `OverlayMapa`) hasta el momento:

```
public class OverlayMapa extends Overlay {
    private Double latitud = 37.40*1E6;
    private Double longitud = -5.99*1E6;

    @Override
    public void draw(Canvas canvas, MapView mapView, boolean shadow)
    {
        Projection projection = mapView.getProjection();
        GeoPoint geoPoint =
            new GeoPoint(latitud.intValue(), longitud.intValue());

        if (shadow == false)
        {
            Point centro = new Point();
            projection.toPixels(geoPoint, centro);

            //Definimos el pincel de dibujo
            Paint p = new Paint();
            p.setColor(Color.BLUE);

            //Marca Ejemplo 1: Círculo y Texto
            canvas.drawCircle(centro.x, centro.y, 5, p);
            canvas.drawText("Sevilla", centro.x+10, centro.y+5, p);
        }
    }
}
```

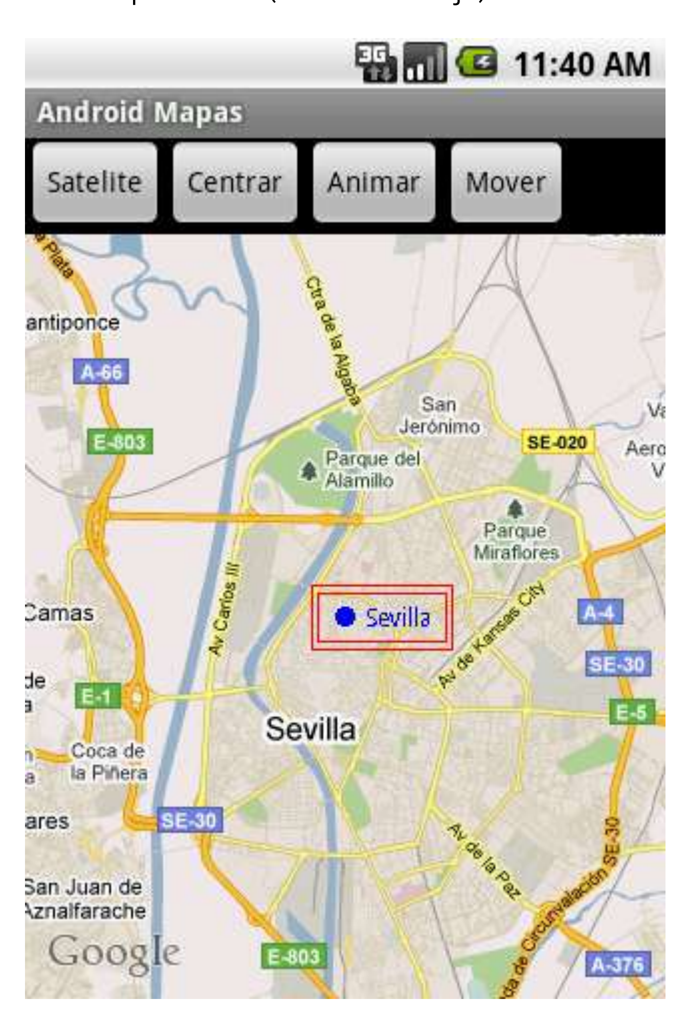
Una vez definida nuestra capa de información personalizada debemos añadirla al mapa que se va a mostrar en la aplicación de ejemplo. En nuestro caso esto lo haremos desde el método `onCreate()` de la actividad principal, tras obtener la referencia al control `MapView`. Para ello, obtendremos la lista de capas del mapa mediante el método `getOverlays()`, crearemos una nueva instancia de nuestra capa personalizada `OverlayMapa`, y la añadiremos a la lista mediante el método `add()`. Finalmente llamaremos al método `postInvalidate()` para redibujar el mapa y todas sus capas.

```
//Obtenemos una referencia a los controles
mapa = (MapView)findViewById(R.id.mapa);

//...

//Añadimos la capa de marcadores
List<Overlay> capas = mapa.getOverlays();
OverlayMapa om = new OverlayMapa();
capas.add(om);
mapa.postInvalidate();
```

Si ejecutamos la aplicación en este momento podremos comprobar cómo se muestra un mapa centrado en nuestras coordenadas y se dibuja correctamente la información de nuestra nueva capa sobre él (lo resalto en rojo).

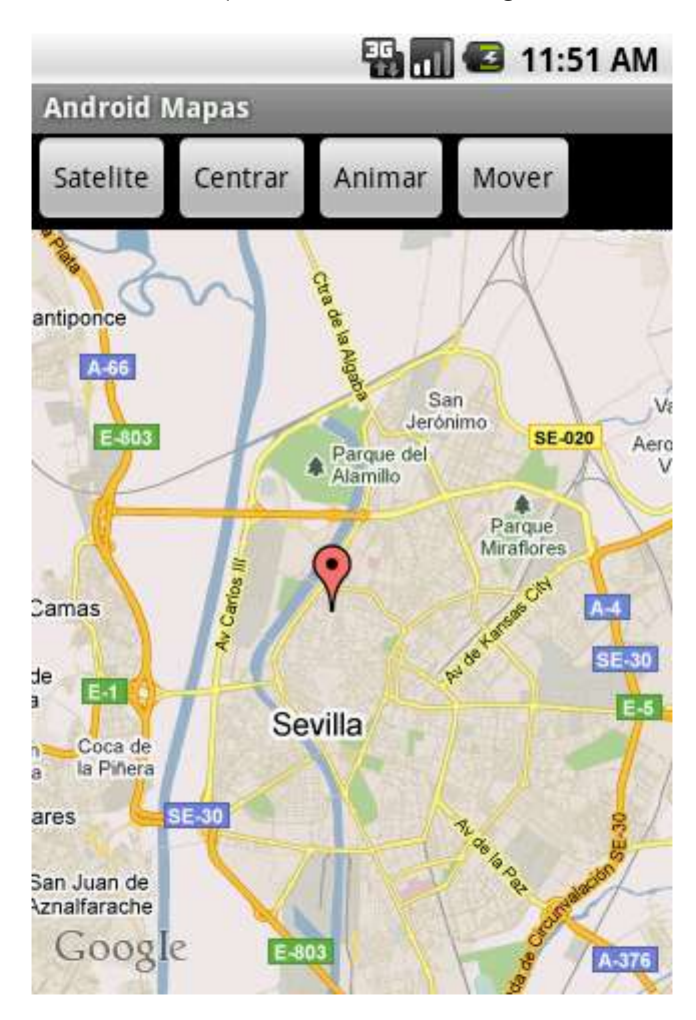


Una posible variante podría ser incluir algún tipo de marcador gráfico sobre el mapa, al estilo del propio *Google Maps*. Para conseguir esto, deberemos colocar la imagen del marcador en las distintas carpetas "`/res/drawable`" y dibujarlo sobre la capa utilizando el método `drawBitmap()`. Para cargar el *bitmap* a partir del fichero de imagen colocado en la carpeta de recursos utilizaremos la clase `BitmapFactory` y su método `decodeResource()`, al que tendremos que pasar como parámetro el ID del recurso. Veamos cómo quedaría esto en el código del método `draw()` de nuestra capa personalizada [para mayor claridad, más abajo podéis descargar como siempre el código fuente completo]:

```
//Marca Ejemplo 2: Bitmap
Bitmap bm = BitmapFactory.decodeResource(
        mapView.getResources(),
        R.drawable.marcador_google_maps);

canvas.drawBitmap(bm, centro.x - bm.getWidth(),
        centro.y - bm.getHeight(), p);
```

En mi caso, como imagen para el marcador he utilizado una similar al que se muestra en *Google Maps*. Si ejecutamos ahora la aplicación, veremos cómo hemos sustituido la marca textual anterior por el nuevo marcador gráfico:



Ya hemos visto lo sencillo que resulta mostrar información personalizada sobre un mapa. Tan sólo nos falta saber cómo podemos también reaccionar ante pulsaciones del usuario sobre nuestro control.

Para conseguir esto, tendremos que sobrescribir también el método `onTap()` de nuestra capa personalizada. Este método nos proporciona directamente como parámetro las coordenadas de latitud y longitud sobre las que el usuario ha pulsado (en forma de objeto `GeoPoint`), con lo que podremos actuar en consecuencia desde nuestra aplicación.

Como ejemplo, nosotros nos vamos a limitar a mostrar en una notificación de tipo `Toast`, las coordenadas sobre las que se ha pulsado.

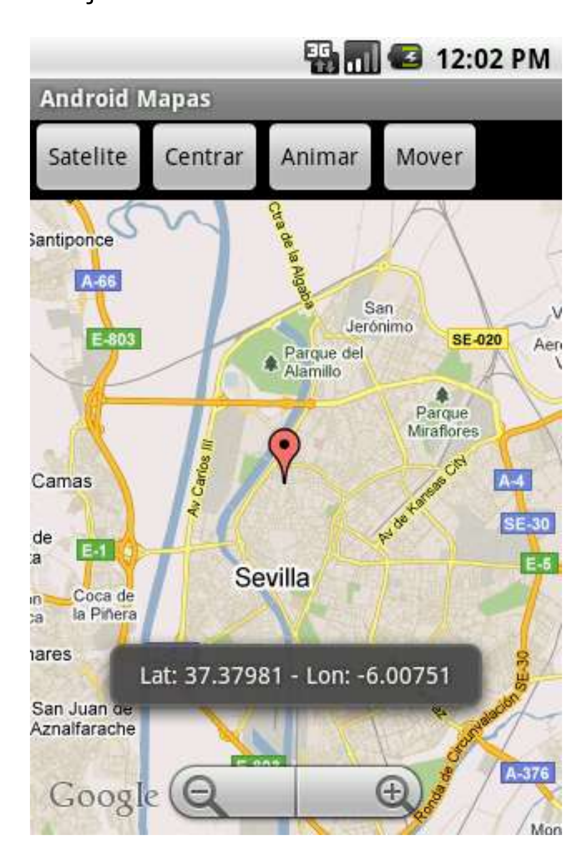
```
@Override
public boolean onTap(GeoPoint point, MapView mapView)
{
    Context contexto = mapView.getContext();
    String msg = "Lat: " + point.getLatitudeE6()/1E6 + " - " +
        "Lon: " + point.getLongitudeE6()/1E6;

    Toast toast = Toast.makeText(contexto, msg, Toast.LENGTH_SHORT);
    toast.show();
```

```
    return true;
}
```

El método debe devolver el valor true siempre que haya tratado la pulsación y NO sea necesario notificarla al resto de capas o al propio control `MapView`, y false en caso contrario.

Si ejecutamos de nuevo la aplicación de ejemplo y probamos a pulsar sobre cualquier lugar del mapa mostrado veremos cómo se muestran las coordenadas que se han seleccionado. Esto se podría utilizar por ejemplo para detectar si se pulsa sobre alguno de nuestros marcadores con el objetivo de mostrar información adicional.



El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-mapas-3.zip



Ficheros

X. Ficheros en Android

Ficheros en Memoria Interna

En apartados anteriores del Curso de Programación Android hemos visto ya diversos métodos para almacenar datos en nuestras aplicaciones, como por ejemplo los ficheros de preferencias compartidas o las bases de datos SQLite. Estos mecanismos son perfectos para almacenar datos estructurados, pero en ocasiones nos seguirá siendo útil poder disponer también de otros ficheros auxiliares de datos, probablemente con otro tipo de contenidos y formatos. Por ello, en Android también podremos manipular ficheros tradicionales de una forma muy similar a como se realiza en Java.

Lo primero que hay que tener en cuenta es dónde queremos almacenar los ficheros y el tipo de acceso que queremos tener a ellos. Así, podremos leer y escribir ficheros localizados en:

- La **memoria interna** del dispositivo.
- La **tarjeta SD** externa, si existe.
- La propia aplicación, en forma de **recurso**.

En los dos próximos apartados aprenderemos a manipular ficheros almacenados en cualquiera de estos lugares, comentando las particularidades de cada caso.

Veamos en primer lugar cómo trabajar con la memoria interna del dispositivo. Cuando almacenamos ficheros en la memoria interna debemos tener en cuenta las limitaciones de espacio que tienen muchos dispositivos, por lo que no deberíamos abusar de este espacio utilizando ficheros de gran tamaño.

Escribir ficheros en la memoria interna es muy sencillo. Android proporciona para ello el método `openFileOutput()`, que recibe como parámetros el nombre del fichero y el modo de acceso con el que queremos abrir el fichero. Este modo de acceso puede variar entre `MODE_PRIVATE` (por defecto) para acceso privado desde nuestra aplicación, `MODE_APPEND` para añadir datos a un fichero ya existente, `MODE_WORLD_READABLE` para permitir a otras aplicaciones leer el fichero, o `MODE_WORLD_WRITABLE` para permitir a otras aplicaciones escribir sobre el fichero.

Este método devuelve una referencia al *stream* de salida asociado al fichero (en forma de objeto `FileOutputStream`), a partir del cual ya podremos utilizar los métodos de manipulación de ficheros tradicionales del lenguaje java (api `java.io`). Como ejemplo, convertiremos este *stream* a un `OutputStreamWriter` para escribir una cadena de texto al fichero.

```
try
{
    OutputStreamWriter fout=
        new OutputStreamWriter(
            openFileOutput("prueba_int.txt", Context.MODE_PRIVATE));

    fout.write("Texto de prueba.");
    fout.close();
}
catch (Exception ex)
{
    Log.e("Ficheros", "Error al escribir fichero a memoria interna");
}
```

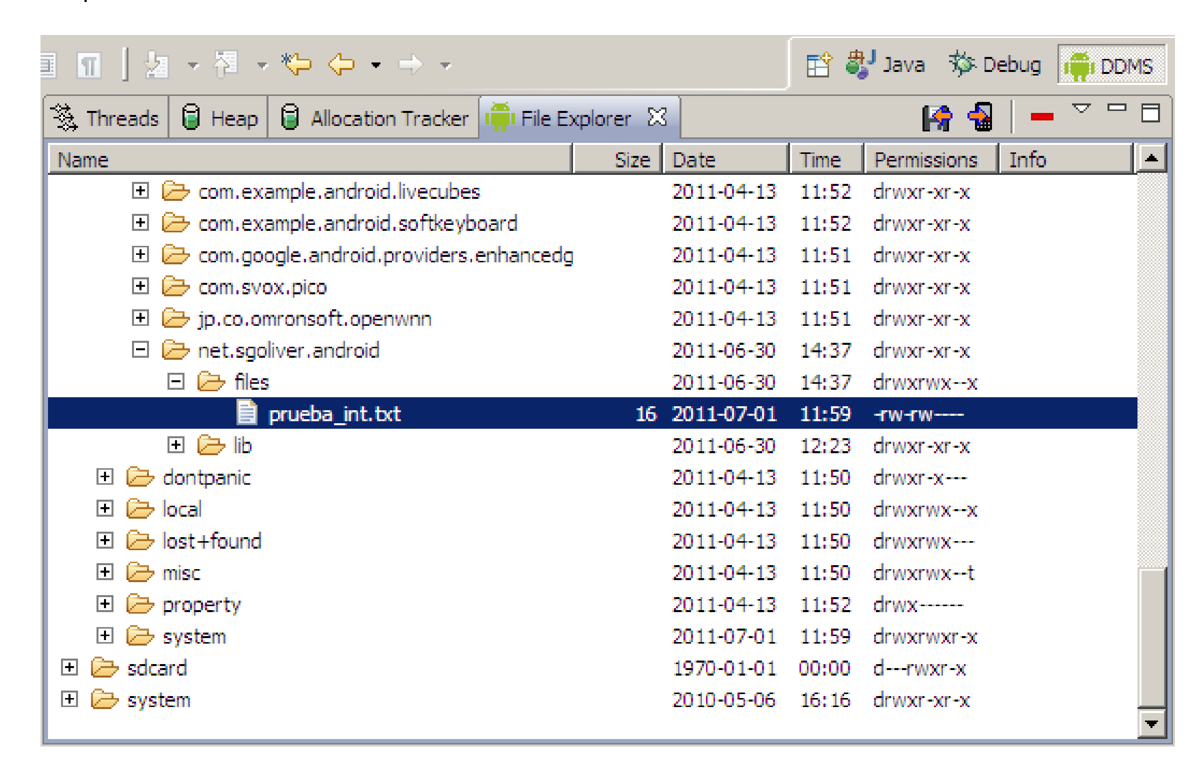
Está bien, ya hemos creado un fichero de texto en la memoria interna, ¿pero dónde exactamente? Tal como ocurría con las bases de datos SQLite, Android almacena por defecto los ficheros creados en una ruta determinada, que en este caso seguirá el siguiente patrón:

`/data/data/paquete_java/files/nombre_fichero`

En mi caso particular, la ruta será

`/data/data/net.sgoliver.android/files/prueba_int.txt`

Si ejecutamos el código anterior podremos comprobar en el DDMS cómo el fichero se crea correctamente en la ruta indicada (Al final del apartado hay un enlace a una aplicación de ejemplo sencilla donde incluyo un botón por cada uno de los puntos que vamos a comentar en el apartado).



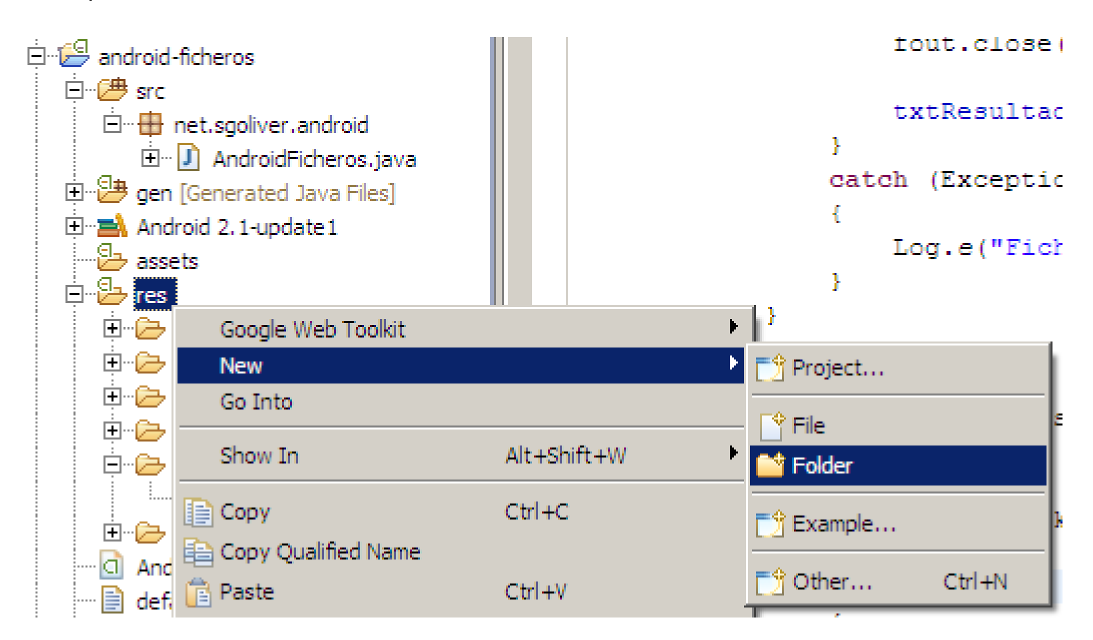
Por otra parte, leer ficheros desde la memoria interna es igual de sencillo, y procederemos de forma análoga, con la única diferencia de que utilizaremos el método `openFileInput()` para abrir el fichero, y los métodos de lectura de java.io para leer el contenido.

```
try
{
    BufferedReader fin =
        new BufferedReader(
            new InputStreamReader(
                openFileInput("prueba_int.txt")));

    String texto = fin.readLine();
    fin.close();
}
catch (Exception ex)
{
    Log.e("Ficheros", "Error al leer fichero desde memoria interna");
}
```

La segunda forma de almacenar ficheros en la memoria interna del dispositivo es incluirlos como recurso en la propia aplicación. Aunque este método es útil en muchos casos, sólo debemos utilizarlo cuando no necesitemos realizar modificaciones sobre los ficheros, ya que tendremos limitado el acceso a sólo lectura.

Para incluir un fichero como recurso de la aplicación debemos colocarlo en la carpeta "`/res/raw`" de nuestro proyecto de Eclipse. Esta carpeta no suele estar creada por defecto, por lo que deberemos crearla manualmente en Eclipse desde el menú contextual con la opción "*New / Folder*".



Una vez creada la carpeta `/raw` podremos colocar en ella cualquier fichero que queramos que se incluya con la aplicación en tiempo de compilación en forma de recurso. Nosotros incluiremos como ejemplo un fichero de texto llamado "`prueba_raw.txt`". Ya en tiempo de ejecución podremos acceder a este fichero, sólo en modo de lectura, de una forma similar a la que ya hemos visto para el resto de ficheros en memoria interna.

Para acceder al fichero, accederemos en primer lugar a los recursos de la aplicación con el método `getResources()` y sobre éstos utilizaremos el método `openRawResource(id_del_recurso)` para abrir el fichero en modo lectura. Este

método devuelve un objeto InputStream, que ya podremos manipular como queramos mediante los métodos de la API java.io. Como ejemplo, nosotros convertiremos el stream en un objeto BufferedReader para leer el texto contenido en el fichero de ejemplo (por supuesto los ficheros de recurso también pueden ser binarios, como por ejemplo ficheros de imagen, video, etc). Veamos cómo quedaría el código:

```
try
{
    InputStream fraw =
        getResources().openRawResource(R.raw.prueba_raw);

    BufferedReader brin =
        new BufferedReader(new InputStreamReader(fraw));

    String linea = brin.readLine();

    fraw.close();
}
catch (Exception ex)
{
    Log.e("Ficheros", "Error al leer fichero desde recurso raw");
}
```

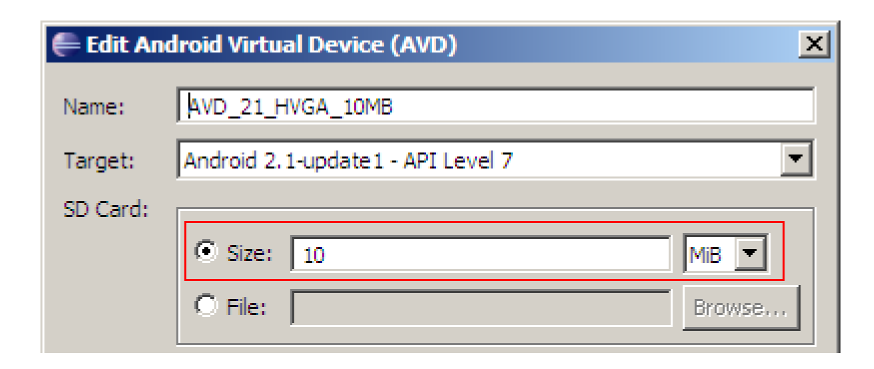
Como puede verse en el código anterior, al método openRawResource() le pasamos como parámetro el ID del fichero incluido como recurso, que seguirá el patrón "R.raw.nombre_del_fichero", por lo que en nuestro caso particular será R.raw.prueba_raw.

Para no alargar mucho el apartado, dejamos la gestión de ficheros en la memoria externa (tarjeta SD) para el próximo apartado, donde también publicaré una sencilla aplicación de ejemplo que incluya toda la funcionalidad que hemos comentado sobre la gestión de ficheros.

Ficheros en Memoria Externa (Tarjeta SD)

En el apartado anterior del curso hemos visto cómo manipular ficheros localizados en la memoria interna de un dispositivo Android. Sin embargo, como ya indicamos, esta memoria suele ser relativamente limitada y no es aconsejable almacenar en ella ficheros de gran tamaño. La alternativa natural es utilizar para ello la memoria externa del dispositivo, constituida normalmente por una tarjeta de memoria SD.

Una nota rápida antes de empezar con este tema. Para poder probar aplicaciones que hagan uso de la memoria externa en el emulador de Android necesitamos tener configurado en Eclipse un AVD que tenga establecido correctamente el tamaño de la tarjeta SD. En mi caso, he definido por ejemplo un tamaño de tarjeta de 10 Mb:



Seguimos. A diferencia de la memoria interna, la tarjeta de memoria no tiene por qué estar presente en el dispositivo, e incluso estándolo puede no estar reconocida por el sistema. Por tanto, el primer paso recomendado a la hora de trabajar con ficheros en memoria externa es asegurarnos de que dicha memoria está presente y disponible para leer y/o escribir en ella.

Para esto la API de Android proporciona (como método estático dela clase `Environment`) el método `getExternalStorageStatus()`, que no dice si la memoria externa está disponible y si se puede leer y escribir en ella. Este método devuelve una serie de valores que nos indicarán el estado de la memoria externa, siendo los más importantes los siguientes:

- `MEDIA_MOUNTED`, que indica que la memoria externa está disponible y podemos tanto leer como escribir en ella.
- `MEDIA_MOUNTED_READ_ONLY`, que indica que la memoria externa está disponible pero sólo podemos leer de ella.
- Otra serie de valores que indicarán que existe algún problema y que por tanto no podemos ni leer ni escribir en la memoria externa (`MEDIA_UNMOUNTED`, `MEDIA_REMOVED`, ...). Podéis consultar todos estos estados en la documentación oficial de la clase `Environment`.

Con todo esto en cuenta, podríamos realizar un chequeo previo del estado de la memoria externa del dispositivo de la siguiente forma:

```
boolean sdDisponible = false;
boolean sdAccesoEscritura = false;

//Comprobamos el estado de la memoria externa (tarjeta SD)
String estado = Environment.getExternalStorageState();

if (estado.equals(Environment.MEDIA_MOUNTED))
{
    sdDisponible = true;
    sdAccesoEscritura = true;
}
else if (estado.equals(Environment.MEDIA_MOUNTED_READ_ONLY))
{
    sdDisponible = true;
    sdAccesoEscritura = false;
}
else
{
    sdDisponible = false;
    sdAccesoEscritura = false;
}
```

Una vez chequeado el estado de la memoria externa, y dependiendo del resultado obtenido, ya podremos leer o escribir en ella cualquier tipo de fichero.

Empecemos por la escritura. Para escribir un fichero a la tarjeta SD tenemos que obtener en primer lugar la ruta al directorio raíz de esta memoria. Para ello utilizaremos el método `getExternalStorageDirectory()` de la clase `Environment`, que nos devolverá un objeto File con la ruta de dicho directorio. A partir de este objeto, podremos construir otro con el nombre elegido para nuestro fichero (como ejemplo "`prueba_sd.txt`"), creando un nuevo objeto `File` que combine ambos elementos. Tras esto, ya sólo queda encapsularlo en algún objeto de escritura de ficheros de la API de java y escribir algún dato de prueba. En nuestro caso de ejemplo lo convertiremos una vez más a un objeto `OutputStreamWriter` para escribir al fichero un mensaje de texto. Veamos cómo quedaría el código:

```
try
{
    File ruta_sd = Environment.getExternalStorageDirectory();

    File f = new File(ruta_sd.getAbsolutePath(), "prueba_sd.txt");

    OutputStreamWriter fout =
        new OutputStreamWriter(
            new FileOutputStream(f));

    fout.write("Texto de prueba.");
    fout.close();
} catch (Exception ex)
{
    Log.e("Ficheros", "Error al escribir fichero a tarjeta SD");
}
```

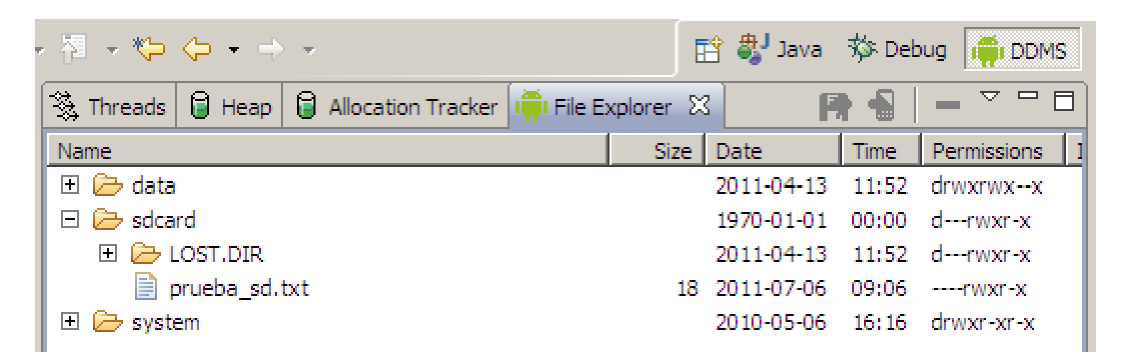
Además de esto, tendremos que especificar en el fichero `AndroidManifest.xml` que nuestra aplicación necesita permiso de escritura en la memoria externa. Para añadir un nuevo permiso usaremos como siempre la cláusula `<uses-permission>` utilizando el valor concreto "`android.permission.WRITE_EXTERNAL_STORAGE`". Con esto, nuestro `AndroidManifest.xml` quedaría de forma similar a éste:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="net.sgoliver.android" android:versionCode="1"
      android:versionName="1.0">
    <uses-sdk android:minSdkVersion="7" />

    <uses-permission
        android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    </uses-permission>

    <application android:icon="@drawable/icon"
                 android:label="@string/app_name">
        <activity android:name=".AndroidFicheros"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Si ejecutamos ahora el código y nos vamos al explorador de archivos del DDMS podremos comprobar cómo se ha creado correctamente el fichero en el directorio raíz de nuestra SD (carpeta `/sdcard/`).



Por su parte, leer un fichero desde la tarjeta SD es igual de sencillo. Obtenemos el directorio raíz de la memoria externa con `getExternalStorageDirectory()`, creamos un objeto `File` que combine esa ruta con el nombre del fichero a leer y lo encapsulamos dentro de algún objeto que facilite la lectura, nosotros para leer texto utilizaremos como siempre un `BufferedReader`.

```
try
{
    File ruta_sd = Environment.getExternalStorageDirectory();

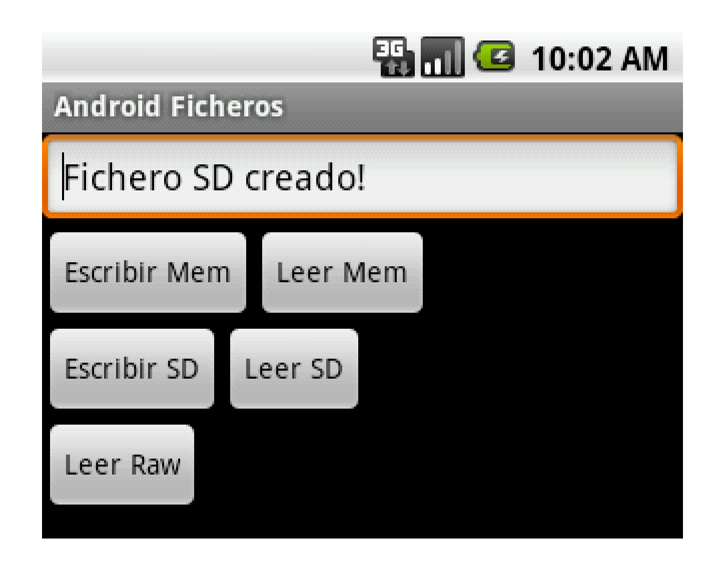
    File f = new File(ruta_sd.getAbsolutePath(), "prueba_sd.txt");

    BufferedReader fin =
        new BufferedReader(
            new InputStreamReader(
                new FileInputStream(f)));

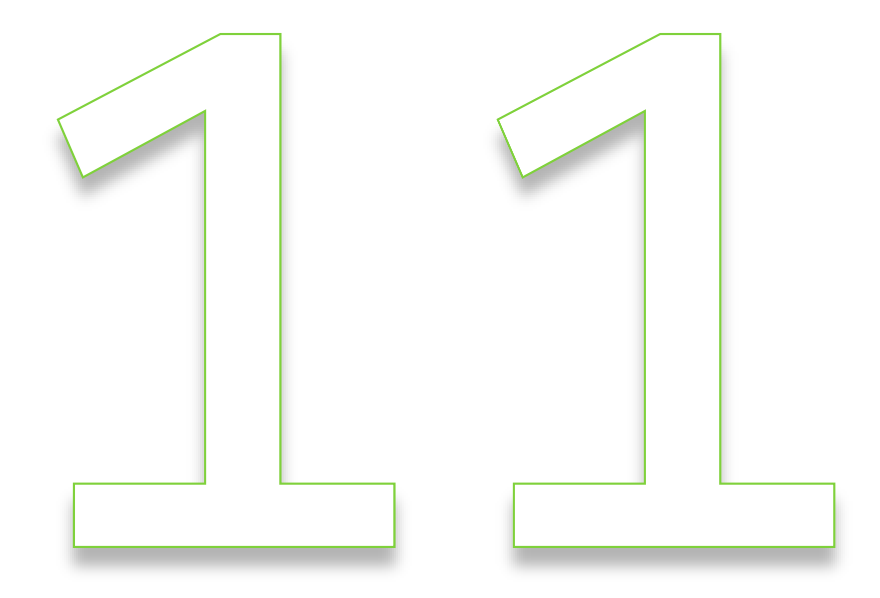
    String texto = fin.readLine();
    fin.close();
}
catch (Exception ex)
{
    Log.e("Ficheros", "Error al leer fichero desde tarjeta SD");
}
```

Como vemos, el código es análogo al que hemos visto para la escritura de ficheros.

Dejé pendiente poneros disponible el código de alguna aplicación sencilla que contenga todos los temas vistos en estos dos últimos apartados sobre gestión de ficheros en Android. Y como lo prometido es deuda, os dejo aquí el enlace al código fuente completo de una aplicación de ejemplo, que simplemente habilita un botón para realizar cada una de las tareas comentadas, mostrando el resultado de cada acción en un cuadro de texto superior de la pantalla principal. Os dejo una imagen para que os hagáis una idea:



El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-ficheros.zip



Content Providers

XI. Content Providers

Construcción de Content Providers

En este nuevo apartado del Curso de Programación en Android que estamos publicando vamos a tratar el temido [o a veces incomprendido] tema de los *Content Providers*.

Un *Content Provider* no es más que el mecanismo proporcionado por la plataforma Android para permitir compartir información entre aplicaciones. Una aplicación que desee que todo o parte de la información que almacena esté disponible de una forma controlada para el resto de aplicaciones del sistema deberá proporcionar un *content provider* a través del cual se pueda realizar el acceso a dicha información. Este mecanismo es utilizado por muchas de las aplicaciones estándar de un dispositivo Android, como por ejemplo la lista de contactos, la aplicación de SMS, o el calendario/agenda. Esto quiere decir que podríamos acceder a los datos gestionados por estas aplicaciones desde nuestras propias aplicaciones Android haciendo uso de los *content providers* correspondientes.

Son por tanto dos temas los que debemos tratar en este apartado, por un lado a construir nuevos *content providers* personalizados para nuestras aplicaciones, y por otro utilizar un *content provider* ya existente para acceder a los datos publicados por otras aplicaciones.

En gran parte de la bibliografía sobre programación en Android se suele tratar primero el tema del acceso a *content providers* ya existentes (como por ejemplo el acceso a la lista de contactos de Android) para después pasar a la construcción de nuevos *content providers* personalizados. Yo sin embargo voy a tratar de hacerlo en orden inverso, ya que me parece importante conocer un poco el funcionamiento interno de un *content provider* antes de pasar a utilizarlo sin más dentro de nuestras aplicaciones. Así, en este primer apartado sobre el tema veremos cómo crear nuestro propio *content provider* para compartir datos con otras aplicaciones, y en el próximo apartado veremos cómo utilizar este mecanismo para acceder directamente a datos de terceros.

Empecemos a entrar en materia. Para añadir un *content provider* a nuestra aplicación tendremos que:

1. Crear una nueva clase que extienda a la clase android `ContentProvider`.
2. Declarar el nuevo *content provider* en nuestro fichero AndroidManifest.xml

Por supuesto nuestra aplicación tendrá que contar previamente con algún método de almacenamiento interno para la información que queremos compartir. Lo más común será disponer de una base de datos SQLite, por lo que será esto lo que utilizaré para todos los ejemplos de este apartado, pero internamente podríamos tener los datos almacenados de cualquier otra forma, por ejemplo en ficheros de texto, ficheros XML, etc. El content provider será el mecanismo que nos permita publicar esos datos a terceros de una forma homogénea y a través de una interfaz estandarizada.

Un primer detalle a tener en cuenta es que los registros de datos proporcionados por un content provider deben contar siempre con un campo llamado _ID que los identifique de forma unívoca del resto de registros. Como ejemplo, los registros devueltos por un content provider de clientes podría tener este aspecto:

_ID	Cliente	Telefono	Email
3	Antonio	900123456	email1@correo.com
7	Jose	900123123	email2@correo.com
9	Luis	900123987	email3@correo.com

Sabiendo esto, es interesante que nuestros datos también cuenten internamente con este campo `_ID` (no tiene por qué llamarse igual) de forma que nos sea más sencillo después generar los resultados del content provider.

Con todo esto, y para tener algo desde lo que partir, vamos a construir en primer lugar una aplicación de ejemplo muy sencilla con una base de datos SQLite que almacene los datos de una serie de clientes con una estructura similar a la tabla anterior. Para ello seguiremos los mismos pasos que ya comentamos en los apartados dedicados al tratamiento de bases de datos SQLite en Android (consultar índice del curso).

Por volver a recordarlo muy brevemente, lo que haremos será crear una nueva clase que extienda a `SQLiteOpenHelper`, definiremos las sentencias SQL para crear nuestra tabla de clientes, e implementaremos finalmente los métodos `onCreate()` y `onUpgrade()`. El código de esta nueva clase, que yo he llamado `ClientesSqliteHelper`, quedaría como sigue:

```
public class ClientesSqliteHelper extends SQLiteOpenHelper {

    //Sentencia SQL para crear la tabla de Clientes
    String sqlCreate = "CREATE TABLE Clientes " +
                       "(_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                       " nombre TEXT, " +
                       " telefono TEXT, " +
                       " email TEXT )";

    public ClientesSqliteHelper(Context contexto, String nombre,
                               CursorFactory factory, int version) {

        super(contexto, nombre, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        //Se ejecuta la sentencia SQL de creación de la tabla
        db.execSQL(sqlCreate);

        //Insertamos 15 clientes de ejemplo
        for(int i=1; i<=15; i++)
        {
            //Generamos los datos de muestra
            String nombre = "Cliente" + i;
            String telefono = "900-123-00" + i;
            String email = "email" + i + "@mail.com";
```

```
            //Insertamos los datos en la tabla Clientes
            db.execSQL("INSERT INTO Clientes (nombre, telefono, email) " +
                       "VALUES ('" + nombre + "', '" + telefono +"', '" +
                       email + "')");
        }
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int versionAnterior,
                          int versionNueva) {

        //NOTA: Por simplicidad del ejemplo aquí utilizamos directamente la
        //      opción de eliminar la tabla anterior y crearla de nuevo
        //      vacía con el nuevo formato.
        //      Sin embargo lo normal será que haya que migrar datos de la
        //      tabla antigua a la nueva, por lo que este método debería
        //      ser más elaborado.

        //Se elimina la versión anterior de la tabla
        db.execSQL("DROP TABLE IF EXISTS Clientes");

        //Se crea la nueva versión de la tabla
        db.execSQL(sqlCreate);
    }
}
```

Como notas relevantes del código anterior:

- Nótese el campo "_id" que hemos incluido en la base de datos de clientes por lo motivos indicados un poco más arriba. Este campo lo declaramos como `INTEGER PRIMARY KEY AUTOINCREMENT`, de forma que se incremente automáticamente cada vez que insertamos un nuevo registro en la base de datos.
- En el método `onCreate()`, además de ejecutar la sentencia SQL para crear la tabla Clientes, también inserta varios registros de ejemplo.
- Para simplificar el ejemplo, el método `onUpgrade()` se limita a eliminar la tabla actual y crear una nueva con la nueva estructura. En una aplicación real habría que hacer probablemente la migración de los datos a la nueva base de datos.

Dado que la clase anterior ya se ocupa de todo, incluso de insertar algunos registro de ejemplo con los que podamos hacer pruebas, la aplicación principal de ejemplo no mostrará en principio nada en pantalla ni hará nada con la información. Esto lo he decidido así para no complicar el código de la aplicación innecesariamente, ya que no nos va a interesar el tratamiento directo de los datos por parte de la aplicación principal, sino su utilización a través del content provider que vamos a construir.

Una vez que ya contamos con nuestra aplicación de ejemplo y su base de datos, es hora de empezar a construir el nuevo content provider que nos permitirá compartir sus datos con otras aplicaciones.

Lo primero que vamos a comentar es la forma con que se hace referencia en Android a los content providers. El acceso a un content provider se realiza siempre mediante una URI. Una URI no es más que una cadena de texto similar a cualquiera de las direcciones web que utilizamos en nuestro navegador. Al igual que para acceder a mi blog lo hacemos mediante la

dirección "http://www.sgoliver.net", para acceder a un content provider utilizaremos una dirección similar a "`content://net.sgoliver.android.ejemplo/clientes`".

Las direcciones URI de los content providers están formadas por 3 partes. En primer lugar el prefijo "`content://`" que indica que dicho recurso deberá ser tratado por un content provider. En segundo lugar se indica el identificador en sí del content provider, también llamado *authority*. Dado que este dato debe ser único es una buena práctica utilizar un authority de tipo "nombre de clase java invertido", por ejemplo en mi caso "`net.sgoliver.android.ejemplo`". Por último, se indica la entidad concreta a la que queremos acceder dentro de los datos que proporciona el content provider. En nuestro caso será simplemente la tabla de "`clientes`", ya que será la única existente, pero dado que un content provider puede contener los datos de varias entidades distintas en este último tramo de la URI habrá que especificarlo. Indicar por último que en una URI se puede hacer referencia directamente a un registro concreto de la entidad seleccionada. Esto se haría indicando al final de la URI el ID de dicho registro. Por ejemplo la uri "`content://net.sgoliver.android.ejemplo/clientes/23`" haría referencia directa al cliente con _ID = 23.

Todo esto es importante ya que será nuestro content provider el encargado de interpretar/parsear la URI completa para determinar los datos que se le están solicitando. Esto lo veremos un poco más adelante.

Sigamos. El siguiente paso será extender a la clase `ContentProvider`. Si echamos un vistazo a los métodos abstractos que tendremos que implementar veremos que tenemos los siguientes:

- onCreate()
- query()
- insert()
- update()
- delete()
- getType()

El primero de ellos nos servirá para inicializar todos los recursos necesarios para el funcionamiento del nuevo content provider. Los cuatro siguientes serán los métodos que permitirán acceder a los datos (consulta, inserción, modificación y eliminación, respectivamente) y por último, el método `getType()` permitirá conocer el tipo de datos devueltos por el content provider (más tarde intentaremos explicar algo mejor esto último).

Además de implementar estos métodos, también definiremos una serie de constantes dentro de nuestra nueva clase provider, que ayudarán posteriormente a su utilización. Veamos esto paso a paso. Vamos a crear una nueva clase `ClientesProvider` que extienda de `ContentProvider`.

Lo primero que vamos a definir es la URI con la que se accederá a nuestro content provider. En mi caso he elegido la siguiente:

```
content://net.sgoliver.android.ejemplo/clientes
```

Además, para seguir la práctica habitual de todos los content providers de Android, encapsularemos además esta dirección en un objeto estático de tipo Uri llamado `CONTENT_URI`.

```
//Definición del CONTENT_URI
private static final String uri =
    "content://net.sgoliver.android.ejemplo/clientes";

public static final Uri CONTENT_URI = Uri.parse(uri);
```

A continuación vamos a definir varias constantes con los nombres de las columnas de los datos proporcionados por nuestro content provider. Como ya dijimos antes existen columnas predefinidas que deben tener todos los content providers, por ejemplo la columna _ID. Estas columnas estándar están definidas en la clase `BaseColumns`, por lo que para añadir las nuevas columnas de nuestro content provider definiremos una clase interna pública tomando como base la clase `BaseColumns` y añadiremos nuestras nuevas columnas.

```
//Clase interna para declarar las constantes de columna
public static final class Clientes implements BaseColumns
{
    private Clientes() {}

    //Nombres de columnas
    public static final String COL_NOMBRE = "nombre";
    public static final String COL_TELEFONO = "telefono";
    public static final String COL_EMAIL = "email";
}
```

Por último, vamos a definir varios atributos privados auxiliares para almacenar el nombre de la base de datos, la versión, y la tabla a la que accederá nuestro content provider.

```
//Base de datos
private ClientesSqliteHelper clidbh;
private static final String BD_NOMBRE = "DBClientes";
private static final int BD_VERSION = 1;
private static final String TABLA_CLIENTES = "Clientes";
```

Como se indicó anteriormente, la primera tarea que nuestro content provider deberá hacer cuando se acceda a él será interpretar la URI utilizada. Para facilitar esta tarea Android proporciona una clase llamada `UriMatcher`, capaz de interpretar determinados patrones en una URI. Esto nos será útil para determinar por ejemplo si una URI hace referencia a una tabla genérica o a un registro concreto a través de su ID. Por ejemplo:

- `content://net.sgoliver.android.ejemplo/clientes` –> Acceso genérico a tabla de clientes
- `content://net.sgoliver.android.ejemplo/clientes/17` –> Acceso directo al cliente con ID = 17

Para conseguir esto definiremos también como miembro de la clase un objeto `UriMatcher` y dos nuevas constantes que representen los dos tipos de URI que hemos indicado: acceso genérico a tabla (lo llamaré `CLIENTES`) o acceso a cliente por ID (lo llamaré `CLIENTES_ID`). A continuación inicializaremos el objeto `UriMatcher` indicándole el formato de ambos tipos

de URI, de forma que pueda diferenciarlos y devolvernos su tipo (una de las dos constantes definidas, CLIENTES o CLIENTES_ID).

```
//UriMatcher
private static final int CLIENTES = 1;
private static final int CLIENTES_ID = 2;
private static final UriMatcher uriMatcher;

//Inicializamos el UriMatcher
static {
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);

    uriMatcher.addURI("net.sgoliver.android.ejemplo",
                      "clientes", CLIENTES);

    uriMatcher.addURI("net.sgoliver.android.ejemplo",
                      "clientes/#", CLIENTES_ID);
}
```

En el código anterior vemos como mediante el método addUri() indicamos el *authority* de nuestra URI, el formato de la entidad que estamos solicitando, y el tipo con el que queremos identificar dicho formato. Más tarde veremos cómo utilizar esto de forma práctica.

Bien, pues ya tenemos definidos todos los miembros necesarios para nuestro nuevo content provider. Ahora toca implementar los métodos comentados anteriormente.

El primero de ellos es onCreate(). En este método nos limitaremos simplemente a inicializar nuestra base de datos, a través de su nombre y versión, y utilizando para ello la clase ClientesSqliteHelper que creamos al principio del apartado.

```
@Override
public boolean onCreate() {

    clidbh = new ClientesSqliteHelper(
            getContext(), BD_NOMBRE, null, BD_VERSION);

    return true;
}
```

La parte interesante llega con el método query(). Este método recibe como parámetros una URI, una lista de nombres de columna, un criterio de selección, una lista de valores para las variables utilizadas en el criterio anterior, y un criterio de ordenación. Todos estos datos son análogos a los que comentamos cuando tratamos la consulta de datos en SQLite para Android, apartado que recomiendo releer si no tenéis muy frescos estos conocimientos. El método query deberá devolver los datos solicitados según la URI indicada y los criterios de selección y ordenación pasados como parámetro. Así, si la URI hace referencia a un cliente concreto por su ID ése deberá ser el único registro devuelto. Si por el contrario es un acceso genérico a la tabla de clientes habrá que realizar la consulta SQL correspondiente a la base de datos respetando los criterios pasados como parámetro.

Para distinguir entre los dos tipos de URI posibles utilizaremos como ya hemos indicado el objeto uriMatcher, utilizando su método match(). Si el tipo devuelto es CLIENTES_ID, es decir, que se trata de un acceso a un cliente concreto, sustituiremos el criterio de selección

por uno que acceda a la tabla de clientes sólo por el ID indicado en la URI. Para obtener este ID utilizaremos el método `getLastPathSegment()` del objeto uri que extrae el último elemento de la URI, en este caso el ID del cliente.

Hecho esto, ya tan sólo queda realizar la consulta a la base de datos mediante el método `query()` de `SQLiteDatabase`. Esto es sencillo ya que los parámetros son análogos a los recibidos en el método `query()` del content provider.

```
@Override
public Cursor query(Uri uri, String[] projection,
    String selection, String[] selectionArgs, String sortOrder) {

    //Si es una consulta a un ID concreto construimos el WHERE
    String where = selection;
    if(uriMatcher.match(uri) == CLIENTES_ID){
                where = "_id=" + uri.getLastPathSegment();
        }

    SQLiteDatabase db = clidbh.getWritableDatabase();

    Cursor c = db.query(TABLA_CLIENTES, projection, where,
                    selectionArgs, null, null, sortOrder);

    return c;
}
```

Como vemos, los resultados se devuelven en forma de objeto `Cursor`, una vez más exactamente igual a como lo hace el método `query()` de `SQLiteDatabase`.

Por su parte, los métodos `update()` y `delete()` son completamente análogos a éste, con la única diferencia de que devuelven el número de registros afectados en vez de un cursor a los resultados. Vemos directamente el código:

```
@Override
public int update(Uri uri, ContentValues values,
              String selection, String[] selectionArgs) {

    int cont;

    //Si es una consulta a un ID concreto construimos el WHERE
    String where = selection;
    if(uriMatcher.match(uri) == CLIENTES_ID){
            where = "_id=" + uri.getLastPathSegment();
    }

    SQLiteDatabase db = clidbh.getWritableDatabase();

    cont = db.update(TABLA_CLIENTES, values, where, selectionArgs);

    return cont;
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {

    int cont;

    //Si es una consulta a un ID concreto construimos el WHERE
    String where = selection;

    if(uriMatcher.match(uri) == CLIENTES_ID){
```

```
        where = "_id=" + uri.getLastPathSegment();
    }

    SQLiteDatabase db = clidbh.getWritableDatabase();

    cont = db.delete(TABLA_CLIENTES, where, selectionArgs);

    return cont;
}
```

El método `insert()` sí es algo diferente, aunque igual de sencillo. La diferencia en este caso radica en que debe devolver la URI que hace referencia al nuevo registro insertado. Para ello, obtendremos el nuevo ID del elemento insertado como resultado del método `insert()` de `SQLiteDatabase`, y posteriormente construiremos la nueva URI mediante el método auxiliar `ContentUris.withAppendedId()` que recibe como parámetro la URI de nuestro content provider y el ID del nuevo elemento.

```
@Override
public Uri insert(Uri uri, ContentValues values) {

    long regId = 1;

    SQLiteDatabase db = clidbh.getWritableDatabase();

    regId = db.insert(TABLA_CLIENTES, null, values);

    Uri newUri = ContentUris.withAppendedId(CONTENT_URI, regId);

    return newUri;
}
```

Por último, tan sólo nos queda implementar el método `getType()`. Este método se utiliza para identificar el tipo de datos que devuelve el content provider. Este tipo de datos se expresará como un *MIME Type*, al igual que hacen los navegadores web para determinar el tipo de datos que están recibiendo tras una petición a un servidor. Identificar el tipo de datos que devuelve un content provider ayudará por ejemplo a Android a determinar qué aplicaciones son capaces de procesar dichos datos.

Una vez más existirán dos tipos MIME distintos para cada entidad del content provider, uno de ellos destinado a cuando se devuelve una lista de registros como resultado, y otro para cuando se devuelve un registro único concreto. De esta forma, seguiremos los siguientes patrones para definir uno y otro tipo de datos:

- "`vnd.android.cursor.item/vnd.xxxxxx`" –> Registro único
- "`vnd.android.cursor.dir/vnd.xxxxxx`" –> Lista de registros

En mi caso de ejemplo, he definido los siguientes tipos:

- "`vnd.android.cursor.item/vnd.sgoliver.cliente`"
- "`vnd.android.cursor.dir/vnd.sgoliver.cliente`"

Con esto en cuenta, la implementación del método `getType()` quedaría como sigue:

```
@Override
public String getType(Uri uri) {

    int match = uriMatcher.match(uri);

    switch (match)
    {
        case CLIENTES:
            return "vnd.android.cursor.dir/vnd.sgoliver.cliente";
        case CLIENTES_ID:
            return "vnd.android.cursor.item/vnd.sgoliver.cliente";
        default:
            return null;
    }
}
```

Como se puede observar, utilizamos una vez más el objeto `UriMatcher` para determinar el tipo de URI que se está solicitando y en función de ésta devolvemos un tipo MIME u otro.

Pues bien, con esto ya hemos completado la implementación del nuevo content provider. Pero aún nos queda un paso más, como indicamos al principio del apartado. Debemos declarar el content provider en nuestro fichero `AndroidManifest.xml` de forma que una vez instalada la aplicación en el dispositivo Android conozca la existencia de dicho recurso. Para ello, bastará insertar un nuevo elemento `<provider>` dentro de `<application>` indicando el nombre del content provider y su *authority*.

```
<application android:icon="@drawable/icon"
        android:label="@string/app_name">

        ...

        <provider android:name="ClientesProvider"
        android:authorities="net.sgoliver.android.ejemplo"/>

</application>
```

Ahora sí hemos completado totalmente la construcción de nuestro nuevo content provider mediante el cual otras aplicaciones del sistema podrán acceder a los datos almacenados por nuestra aplicación.

En el siguiente apartado veremos cómo utilizar este nuevo content provider para acceder a los datos de nuestra aplicación de ejemplo, y también veremos cómo podemos utilizar alguno de los content provider predefinidos por Android para consultar datos del sistema, como por ejemplo la lista de contactos o la lista de llamadas realizadas.

El código fuente completo de la aplicación lo publicaré junto al siguiente apartado, pero por el momento podéis descargar desde este enlace los fuentes de las dos clases implementadas en este apartado y el fichero `AndroidManifest.xml` modificado.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/Android-content-providers-1.zip

Utilización de Content Providers

En el apartado anterior del Curso de Programación en Android vimos cómo construir un content provider personalizado para permitir a nuestras aplicaciones Android compartir datos con otras aplicaciones del sistema. En este nuevo apartado vamos a ver el tema desde el punto de vista opuesto, es decir, vamos a aprender a hacer uso de un content provider ya existente para acceder a datos de otras aplicaciones. Además, veremos cómo también podemos acceder a datos del propio sistema Android (logs de llamadas, lista de contactos, agenda telefónica, bandeja de entrada de sms, etc) utilizando este mismo mecanismo.

Vamos a comenzar explicando cómo podemos utilizar el content provider que implementamos en el apartado anterior para acceder a los datos de los clientes. Para no complicar mucho el ejemplo ni hacer más difícil las pruebas y la depuración en el emulador de Android vamos a hacer uso el content provider desde la propia aplicación de ejemplo que hemos creado. De cualquier forma, el código necesario sería exactamente igual si lo hiciéramos desde otra aplicación distinta.

Utilizar un content provider ya existente es muy sencillo, sobre todo comparado con el laborioso proceso de construcción de uno nuevo. Para comenzar, debemos obtener una referencia a un Content Resolver, objeto a través del que realizaremos todas las acciones necesarias sobre el content provider. Esto es tan fácil como utilizar el método `getContentResolver()` desde nuestra actividad para obtener la referencia indicada. Una vez obtenida la referencia al content resolver, podremos utilizar sus métodos `query()`, `update()`, `insert()` y `delete()` para realizar las acciones equivalentes sobre el content provider. Por ver varios ejemplos de la utilización de estos métodos añadiremos a nuestra aplicación de ejemplo tres botones en la pantalla principal, uno para hacer una consulta de todos los clientes, otro para insertar registros nuevos, y el último para eliminar todos los registros nuevos insertados con el segundo botón.

Empecemos por la consulta de clientes. El procedimiento será prácticamente igual al que vimos en los apartados de acceso a bases de datos SQLite (consultar el índice del curso). Comenzaremos por definir un *array* con los nombres de las columnas de la tabla que queremos recuperar en los resultados de la consulta, que en nuestro caso serán el ID, el nombre, el teléfono y el email. Tras esto, obtendremos como dijimos antes una referencia al content resolver y utilizaremos su método `query()` para obtener los resultados en forma de cursor. El método `query()` recibe, como ya vimos en el apartado anterior, la Uri del content provider al que queremos acceder, el array de columnas que queremos recuperar, el criterio de selección, los argumentos variables, y el criterio de ordenación de los resultados. En nuestro caso, para no complicarnos utilizaremos tan sólo los dos primeros, pasándole el `CONTENT_URI` de nuestro provider y el array de columnas que acabamos de definir.

```
//Columnas de la tabla a recuperar
String[] projection = new String[] {
    Clientes._ID,
    Clientes.COL_NOMBRE,
    Clientes.COL_TELEFONO,
    Clientes.COL_EMAIL };
```

```
Uri clientesUri =  ClientesProvider.CONTENT_URI;

ContentResolver cr = getContentResolver();

//Hacemos la consulta
Cursor cur = cr.query(clientesUri,
        projection, //Columnas a devolver
        null,       //Condición de la query
        null,       //Argumentos variables de la query
        null);      //Orden de los resultados
```

Hecho esto, tendremos que recorrer el cursor para procesar los resultados. Para nuestro ejemplo, simplemente los escribiremos en un cuadro de texto (`txtResultados`) colocado bajo los tres botones de ejemplo. Una vez más, si tienes dudas sobre cómo recorrer un cursor, puedes consultar los apartados del curso dedicados al tratamiento de bases de datos SQLite, por ejemplo éste. Veamos cómo quedaría el código:

```
if (cur.moveToFirst())
{
    String nombre;
    String telefono;
    String email;

    int colNombre = cur.getColumnIndex(Clientes.COL_NOMBRE);
    int colTelefono = cur.getColumnIndex(Clientes.COL_TELEFONO);
    int colEmail = cur.getColumnIndex(Clientes.COL_EMAIL);

    txtResultados.setText("");

    do
    {
        nombre = cur.getString(colNombre);
        telefono = cur.getString(colTelefono);
        email = cur.getString(colEmail);

        txtResultados.append(nombre + " - " + telefono + " - " + email +
"\n");

    } while (cur.moveToNext());
}
```

Para insertar nuevos registros, el trabajo será también exactamente igual al que se hace al tratar directamente con bases de datos SQLite. Rellenaremos en primer lugar un objeto `ContentValues` con los datos del nuevo cliente y posteriormente utilizamos el método `insert()` pasándole la URI del content provider en primer lugar, y los datos del nuevo registro como segundo parámetro.

```
ContentValues values = new ContentValues();

values.put(Clientes.COL_NOMBRE, "ClienteN");
values.put(Clientes.COL_TELEFONO, "999111222");
values.put(Clientes.COL_EMAIL, "nuevo@email.com");

ContentResolver cr = getContentResolver();

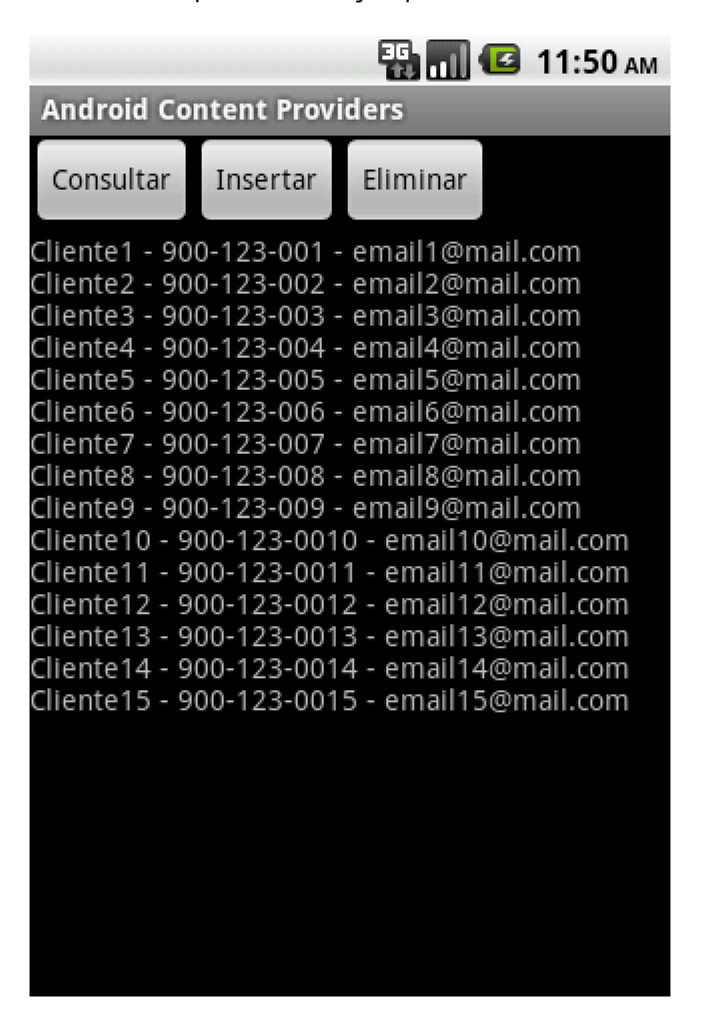
cr.insert(ClientesProvider.CONTENT_URI, values);
```

Por último, y más sencillo todavía, la eliminación de registros la haremos directamente utilizando el método `delete()` del content resolver, indicando como segundo parámetro el criterio de localización de los registros que queremos eliminar, que en este caso serán los que hayamos insertado nuevos con el segundo botón de ejemplo (aquellos con nombre = 'ClienteN').

```
ContentResolver cr = getContentResolver();

cr.delete(ClientesProvider.CONTENT_URI,
        Clientes.COL_NOMBRE + " = 'ClienteN'", null);
```

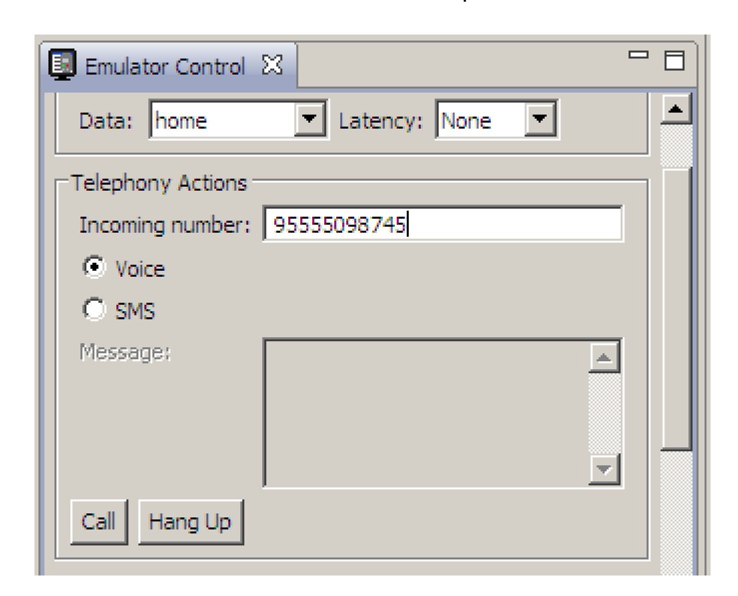
Como muestra gráfica, veamos por ejemplo el resultado de la consulta de clientes (primer botón) en la aplicación de ejemplo.



Con esto, hemos visto lo sencillo que resulta acceder a los datos proporcionados por un content provider. Pues bien, éste es el mismo mecanismo que podemos utilizar para acceder a muchos datos de la propia plataforma Android. En la documentación oficial del paquete android.provider podemos consultar los datos que tenemos disponibles a través de este mecanismo, entre ellos encontramos por ejemplo: el historial de llamadas, la agenda de contactos y teléfonos, las bibliotecas multimedia (audio y video), o el historial y la lista de favoritos del navegador.

Por ver un ejemplo de acceso a este tipo de datos, vamos a realizar una consulta al historial de llamadas del dispositivo, para lo que accederemos al content provider `android.provider.CallLog`.

En primer lugar vamos a registrar varias llamadas en el emulador de Android, de forma que los resultados de la consulta al historial de llamadas contenga algunos registros. Haremos por ejemplo varias llamadas salientes desde el emulador y simularemos varias llamadas entrantes desde el DDMS. Las primeras son sencillas, simplemente ve al emulador, accede al teléfono, marca y descuelga igual que lo harías en un dispositivo físico. Y para emular llamadas entrantes podremos hacerlo una vez más desde Eclipse, accediendo a la vista del DDMS. En esta vista, si accedemos a la sección "*Emulator Control*" veremos un apartado llamado "*Telephony Actions*". Desde éste, podemos introducir un número de teléfono origen cualquiera y pulsar el botón "*Call*" para conseguir que nuestro emulador reciba una llamada entrante. Sin aceptar la llamada en el emulador pulsaremos "*Hang Up*" para terminar la llamada simulando así una llamada perdida.



Hecho esto, procedemos a realizar la consulta al historial de llamadas utilizando el content provider indicado, y para ello añadiremos un botón más a la aplicación de ejemplo.

Consultando la documentación del content provider veremos que podemos extraer diferentes datos relacionados con la lista de llamadas. Nosotros nos quedaremos sólo con dos significativos, el número origen o destino de la llamada, y el tipo de llamada (entrante, saliente, perdida). Los nombres de estas columnas se almacenan en las constantes `Calls.NUMBER` y `Calls.TYPE` respectivamente.

Decidido esto, actuaremos igual que antes. Definiremos el array con las columnas que queremos recuperar, obtendremos la referencia al content resolver y ejecutaremos la consulta llamando al método `query()`. Por último, recorremos el cursor obtenido y procesamos los resultados. Igual que antes, lo único que haremos será escribir los resultados al cuadro de texto situado bajo los botones. Veamos el código:

```
String[] projection = new String[] {
    Calls.TYPE,
    Calls.NUMBER };

Uri llamadasUri =  Calls.CONTENT_URI;

ContentResolver cr = getContentResolver();

Cursor cur = cr.query(llamadasUri,
        projection, //Columnas a devolver
        null,       //Condición de la query
        null,       //Argumentos variables de la query
        null);      //Orden de los resultados

if (cur.moveToFirst())
{
    int tipo;
    String tipoLlamada = "";
    String telefono;

    int colTipo = cur.getColumnIndex(Calls.TYPE);
    int colTelefono = cur.getColumnIndex(Calls.NUMBER);

    txtResultados.setText("");

    do
    {
        tipo = cur.getInt(colTipo);
        telefono = cur.getString(colTelefono);

        if(tipo == Calls.INCOMING_TYPE)
            tipoLlamada = "ENTRADA";
        else if(tipo == Calls.OUTGOING_TYPE)
            tipoLlamada = "SALIDA";
        else if(tipo == Calls.MISSED_TYPE)
            tipoLlamada = "PERDIDA";

        txtResultados.append(tipoLlamada + " - " + telefono + "\n");

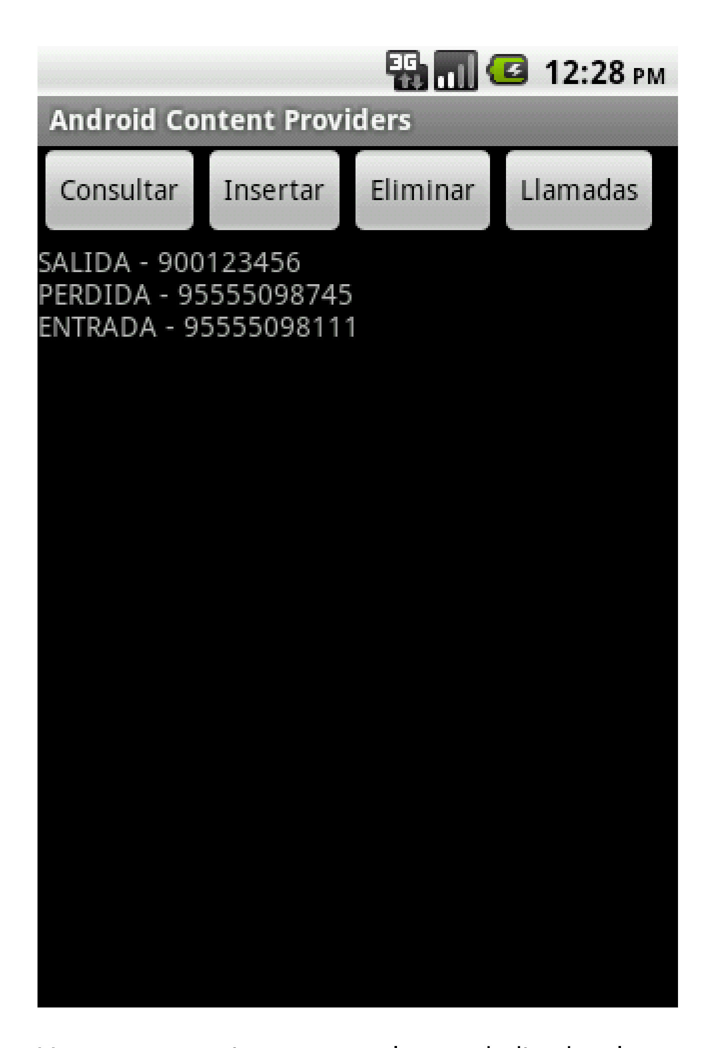
    } while (cur.moveToNext());
}
```

Lo único fuera de lo normal que hacemos en el código anterior es la decodificación del valor del tipo de llamada recuperado, que la hacemos comparando el resultado con las constantes `Calls.INCOMING_TYPE` (entrante), `Calls.OUTGOING_TYPE` (saliente), `Calls.MISSED_TYPE` (perdida) proporcionadas por la propia clase provider.

Un último detalle importante. Para que nuestra aplicación pueda acceder al historial de llamadas del dispositivo tendremos que incluir en el fichero `AndroidManifest.xml` el permiso `READ_CONTACTS` utilizando la cláusula `<uses-permission>` correspondiente.

```
<uses-permission android:name="android.permission.READ_CONTACTS"></uses-
permission>
```

Si ejecutamos la aplicación y realizamos la consulta podremos ver un resultado similar al siguiente:



Y con esto terminamos con el tema dedicado a los *content providers*. Espero que os haya sido útil para aprender a incluir esta funcionalidad a vuestras aplicaciones y a utilizar este mecanismo para acceder a datos propios del sistema.

Podéis descargar el código fuente completo de la aplicación de ejemplo construida a lo largo de los dos apartados anteriores pulsando este enlace.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-content-providers-2.zip



Notificaciones

XII. Notificaciones Android

Notificaciones Toast

En Android existen varias formas de notificar mensajes al usuario, como por ejemplo los cuadros de diálogo modales o las notificaciones de la bandeja del sistema (o barra de estado). Pero en este apartado nos vamos a centrar en primer lugar en la forma más sencilla de notificación: los llamados *Toast*.

Un *toast* es un mensaje que se muestra en pantalla durante unos segundos al usuario para luego volver a desaparecer automáticamente sin requerir ningún tipo de actuación por su parte, y sin recibir el foco en ningún momento (o dicho de otra forma, sin interferir en las acciones que esté realizando el usuario en ese momento). Aunque son personalizables, aparecen por defecto en la parte inferior de la pantalla, sobre un rectángulo gris ligeramente translúcido. Por sus propias características, este tipo de notificaciones son ideales para mostrar mensajes rápidos y sencillos al usuario, pero por el contrario, al no requerir confirmación por su parte, no deberían utilizarse para hacer notificaciones demasiado importantes.

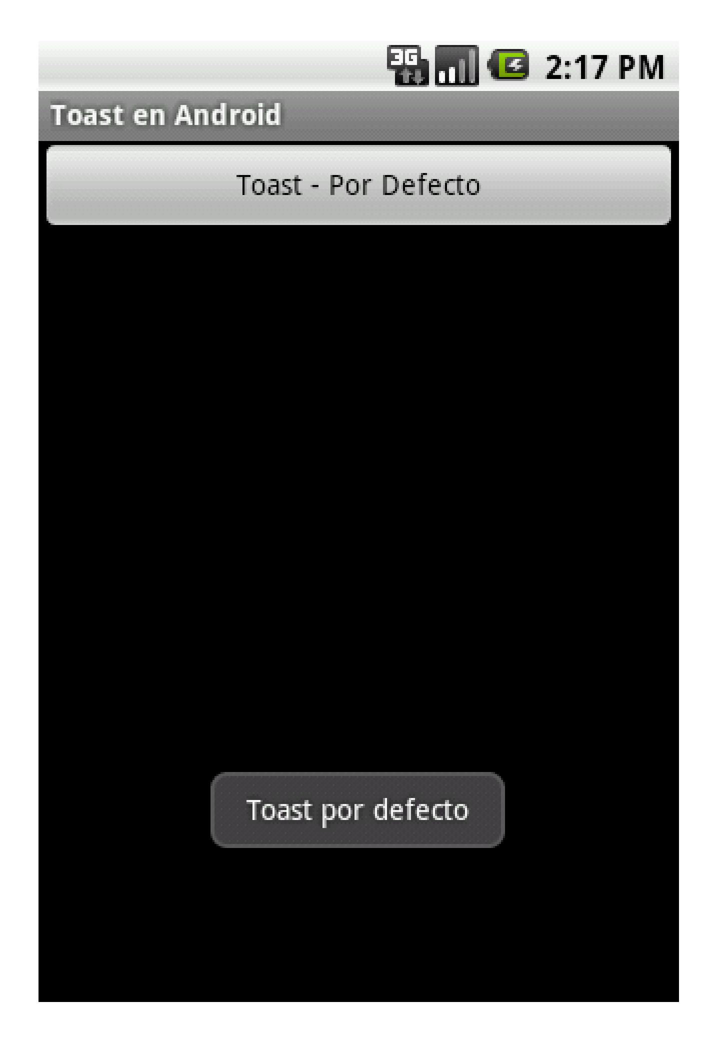
Su utilización es muy sencilla, concentrándose toda la funcionalidad en la clase `Toast`. Esta clase dispone de un método estático `makeText()` al que deberemos pasar como parámetro el contexto de la actividad, el texto a mostrar, y la duración del mensaje, que puede tomar los valores `LENGTH_LONG` o `LENGTH_SHORT`, dependiendo del tiempo que queramos que la notificación aparezca en pantalla. Tras obtener una referencia al objeto `Toast` a través de este método, ya sólo nos quedaría mostrarlo en pantalla mediante el método `show()`.

Vamos a construir una aplicación de ejemplo para demostrar el funcionamiento de este tipo de notificaciones. Y para empezar vamos a incluir un botón que muestre un toast básico de la forma que acabamos de describir:

```
btnDefecto.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        Toast toast1 =
            Toast.makeText(getApplicationContext(),
                    "Toast por defecto", Toast.LENGTH_SHORT);

        toast1.show();
    }
});
```

Si ejecutamos esta sencilla aplicación en el emulador y pulsamos el botón que acabamos de añadir veremos como en la parte inferior de la pantalla aparece el mensaje "`Toast por defecto`", que tras varios segundos desaparecerá automáticamente.



Como hemos comentado, éste es el comportamiento por defecto de las notificaciones toast, sin embargo también podemos personalizarlo un poco cambiando su posición en la pantalla. Para esto utilizaremos el método `setGravity()`, al que podremos indicar en qué zona deseamos que aparezca la notificación. La zona deberemos indicarla con alguna de las constantes definidas en la clase `Gravity`: `CENTER`, `LEFT`, `BOTTOM`, … o con alguna combinación de éstas.

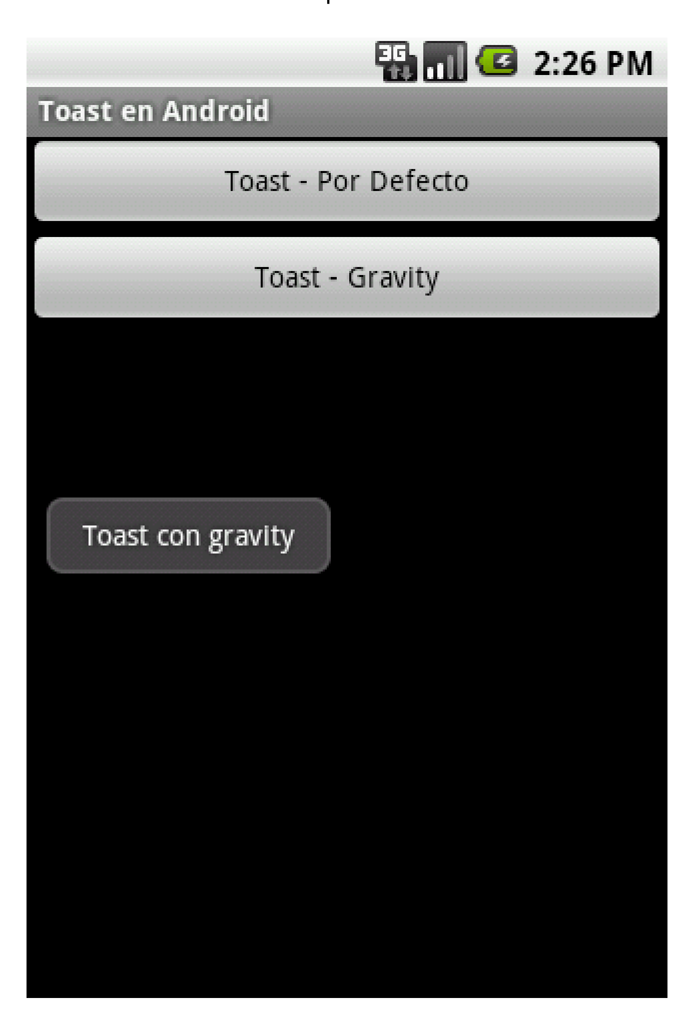
Para nuestro ejemplo vamos a colocar la notificación en la zona central izquierda de la pantalla. Para ello, añadamos un segundo botón a la aplicación de ejemplo que muestre un toast con estas características:

```
btnGravity.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        Toast toast2 =
            Toast.makeText(getApplicationContext(),
                    "Toast con gravity", Toast.LENGTH_SHORT);

        toast2.setGravity(Gravity.CENTER|Gravity.LEFT,0,0);

        toast2.show();
    }
});
```

Si volvemos a ejecutar la aplicación y pulsamos el nuevo botón veremos como el toast aparece en la zona indicada de la pantalla:



Si esto no es suficiente y necesitamos personalizar por completo el aspecto de la notificación, Android nos ofrece la posibilidad de definir un layout XML propio para toast, donde podremos incluir todos los elementos necesarios para adaptar la notificación a nuestras necesidades. Para nuestro ejemplo vamos a definir un layout sencillo, con una imagen y una etiqueta de texto sobre un rectángulo gris:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/lytLayout"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="horizontal"
  android:background="#555555"
  android:padding="5dip" >

    <ImageView android:id="@+id/imgIcono"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:src="@drawable/marcador" />

    <TextView android:id="@+id/txtMensaje"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
```

```
        android:textColor="#FFFFFF"
        android:paddingLeft="10dip" />

</LinearLayout>
```

Guardaremos este layout con el nombre "`toast_layout.xml`", y como siempre lo colocaremos en la carpeta "`res\layout`" de nuestro proyecto.

Para asignar este layout a nuestro toast tendremos qeu actuar de una forma algo diferente a las anteriores. En primer lugar deberemos inflar el layout mediante un objeto `LayoutInflater`, como ya vimos en varias ocasiones al tratar los apartados de interfaz gráfica. Una vez construido el layout modificaremos los valores de los distintos controles para mostrar la información que queramos. En nuestro caso, tan sólo modificaremos el mensaje de la etiqueta de texto, ya que la imagen ya la asignamos de forma estática en el layout XML mediante el atributo `android:src`. Tras esto, sólo nos quedará establecer la duración de la notificación con `setDuration()` y asignar el layout personalizado al toast mediante el método `setView()`. Veamos cómo quedaría todo el código incluido en un tercer botón de ejemplo:

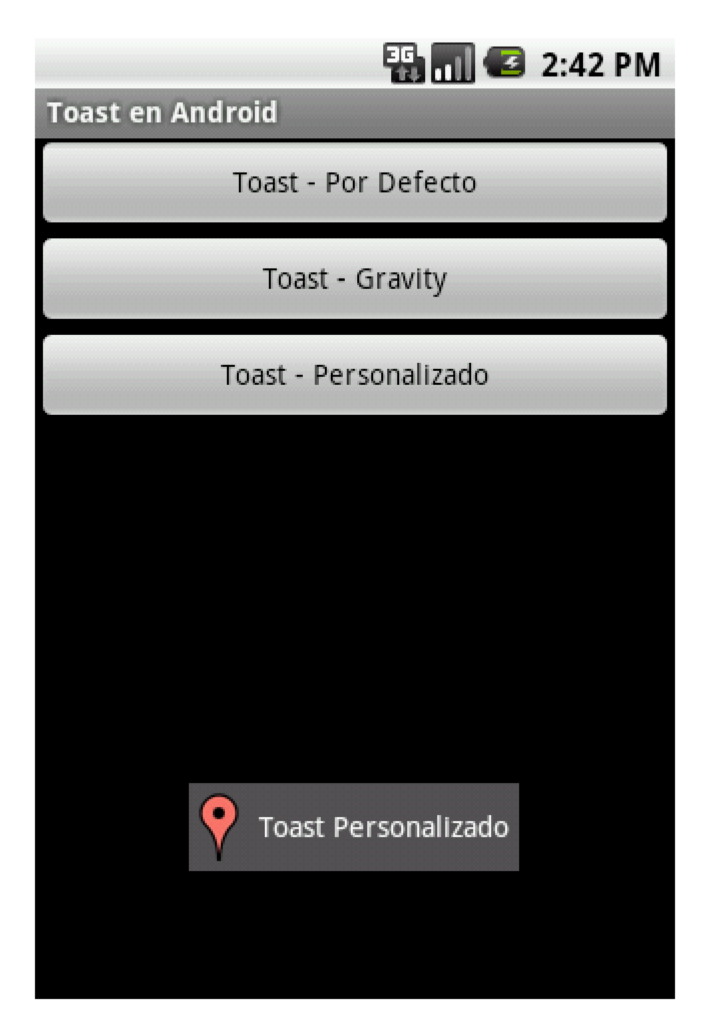
```
btnLayout.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        Toast toast3 = new Toast(getApplicationContext());

        LayoutInflater inflater = getLayoutInflater();
        View layout = inflater.inflate(R.layout.toast_layout,
                        (ViewGroup) findViewById(R.id.lytLayout));

        TextView txtMsg = (TextView)layout.findViewById(R.id.txtMensaje);
        txtMsg.setText("Toast Personalizado");

        toast3.setDuration(Toast.LENGTH_SHORT);
        toast3.setView(layout);
        toast3.show();
    }
});
```

Si ejecutamos ahora la aplicación de ejemplo y pulsamos el nuevo botón, veremos como nuestro toast aparece con la estructura definida en nuestro layout personalizado.



Como podéis comprobar, mostrar notificaciones de tipo Toast en nuestras aplicaciones Android es algo de lo más sencillo, y a veces resultan un elemento de lo más interesante para avisar al usuario de determinados eventos.

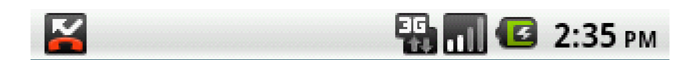
El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-toast.zip

Notificaciones de la Barra de Estado

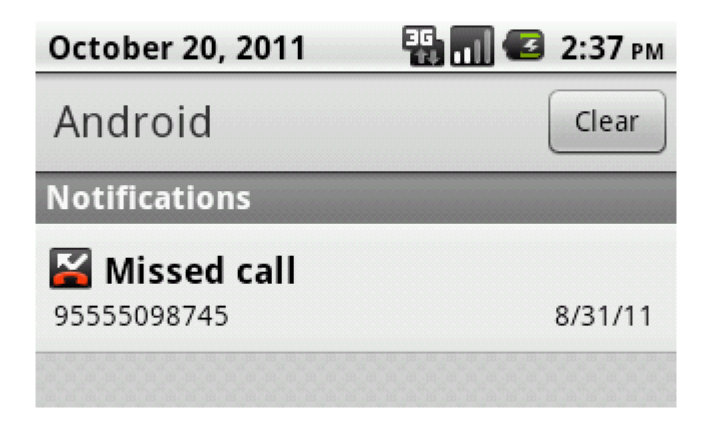
Hace algún tiempo, ya tratamos dentro de este curso un primer mecanismo de notificaciones disponibles en la plataforma Android: los llamados *Toast*. Como ya comentamos, este tipo de notificaciones, aunque resultan útiles y prácticas en muchas ocasiones, no deberíamos utilizarlas en situaciones en las que necesitemos asegurarnos la atención del usuario ya que no requiere de ninguna intervención por su parte, se muestran y desaparecen automáticamente de la pantalla.

En este nuevo apartado vamos a tratar otro tipo de notificaciones algo más persistentes, las notificaciones de la barra de estado de Android. Estas notificaciones son las que se muestran en nuestro dispositivo cuando recibimos un mensaje SMS, cuando tenemos actualizaciones disponibles, cuando tenemos el reproductor de música abierto en segundo plano, … Estas notificaciones constan de un icono y un texto mostrado en la barra de estado superior, y

adicionalmente un mensaje algo más descriptivo y una marca de fecha/hora que podemos consultar desplegando la bandeja del sistema. A modo de ejemplo, cuando tenemos una llamada perdida en nuestro terminal, se nos muestra por un lado un icono en la barra de estado…



… y un mensaje con más información al desplegar la bandeja del sistema, donde en este caso concreto se nos informa del evento que se ha producido ("*Missed call*"), el número de teléfono asociado, y la fecha/hora del evento. Además, al pulsar sobre la notificación se nos dirige automáticamente al historial de llamadas.



Pues bien, aprendamos a utilizar este tipo de notificaciones en nuestras aplicaciones. Vamos a construir para ello una aplicación de ejemplo, como siempre lo más sencilla posible para centrar la atención en lo realmente importante. En este caso, el ejemplo va a consistir en un único botón que genere una notificación de ejemplo en la barra de estado, con todos los elementos comentados y con la posibilidad de dirigirnos a la propia aplicación de ejemplo cuando se pulse sobre ella.

Para generar notificaciones en la barra de estado del sistema, lo primero que debemos hacer es obtener una referencia al servicio de notificaciones de Android, a través de la clase `NotificationManager`. Utilizaremos para ello el método `getSystemService()` indicando como parámetro el identificador del servicio correspondiente, en este caso `Context.NOTIFICATION_SERVICE`.

```
//Obtenemos una referencia al servicio de notificaciones
String ns = Context.NOTIFICATION_SERVICE;
NotificationManager notManager =
    (NotificationManager) getSystemService(ns);
```

Seguidamente configuraremos las características de nuestra notificación. En primer lugar estableceremos el icono y el texto a mostrar en la barra de estado, y registraremos la fecha/hora asociada a nuestra notificación. Con estos datos construiremos un objeto Notification. En nuestro caso de ejemplo, vamos a utilizar un icono predefinido de Android (podéis utilizar cualquier otro), el mensaje de ejemplo "*Alerta!*", y registraremos la fecha/hora actual, devuelta por el método `System.currentTimeMillis()`:

```
//Configuramos la notificación
int icono = android.R.drawable.stat_sys_warning;
CharSequence textoEstado = "Alerta!";
long hora = System.currentTimeMillis();

Notification notif =
    new Notification(icono, textoEstado, hora);
```

El segundo paso será utilizar el método `setLatestEventInfo()` para asociar a nuestra notificación la información a mostrar al desplegar la bandeja del sistema (título y descripción) e indicar la actividad a la cual debemos dirigir al usuario automáticamente si éste pulsa sobre la notificación. Los dos primeros datos son simples cadenas de texto, pero ¿cómo indicamos la aplicación a ejecutar si se pulsa sobre la notificación?

Para esto último debemos construir un objeto `PendingIntent`, que será el que contenga la información de la actividad asociada a la notificación y que será lanzado al pulsar sobre ella. Para ello definiremos en primer lugar un objeto `Intent`, indicando la clase de la actividad concreta a lanzar, que en nuestro caso será la propia actividad principal de ejemplo (`AndroidNotificacionesActivity.class`). Este *intent* lo utilizaremos para construir el `PendingIntent` final mediante el método `PendingIntent.getActivity()`. Veamos cómo quedaría esta última parte comentada:

```
//Configuramos el Intent
Context contexto = getApplicationContext();
CharSequence titulo = "Mensaje de Alerta";
CharSequence descripcion = "Ejemplo de notificación.";

Intent notIntent = new Intent(contexto,
    AndroidNotificacionesActivity.class);

PendingIntent contIntent = PendingIntent.getActivity(
    contexto, 0, notIntent, 0);

notif.setLatestEventInfo(
    contexto, titulo, descripcion, contIntent);
```

Como opciones adicionales, también podemos indicar por ejemplo que nuestra notificación desaparezca automáticamente de la bandeja del sistema cuando se pulsa sobre ella. Esto lo hacemos añadiendo al atributo flags de nuestra notificación el valor `Notification.FLAG_AUTO_CANCEL`.

También podríamos indicar que al generarse la notificación el dispositivo debe emitir un sonido, vibrar o encender el LED de estado presente en muchos terminales. Esto lo conseguiríamos añadiendo al atributo defaults de la notificación los valores `DEFAULT_SOUND`, `DEFAULT_VIBRATE` o `DEFAULT_LIGHTS`.

```
//AutoCancel: cuando se pulsa la notificaión ésta desaparece
notif.flags |= Notification.FLAG_AUTO_CANCEL;

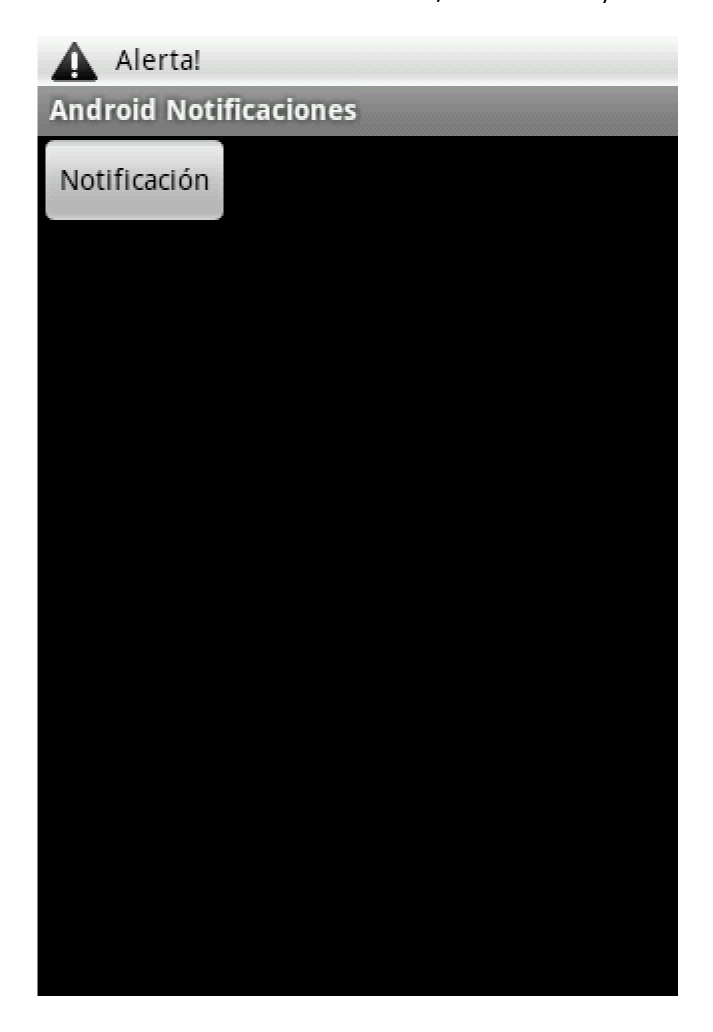
//Añadir sonido, vibración y luces
//notif.defaults |= Notification.DEFAULT_SOUND;
//notif.defaults |= Notification.DEFAULT_VIBRATE;
//notif.defaults |= Notification.DEFAULT_LIGHTS;
```

Existen otras muchas opciones y personalizaciones de éstas últimas, que se pueden consultar en la documentación oficial de la clase `Notification` de Android (atributos `flags` y `defaults`).

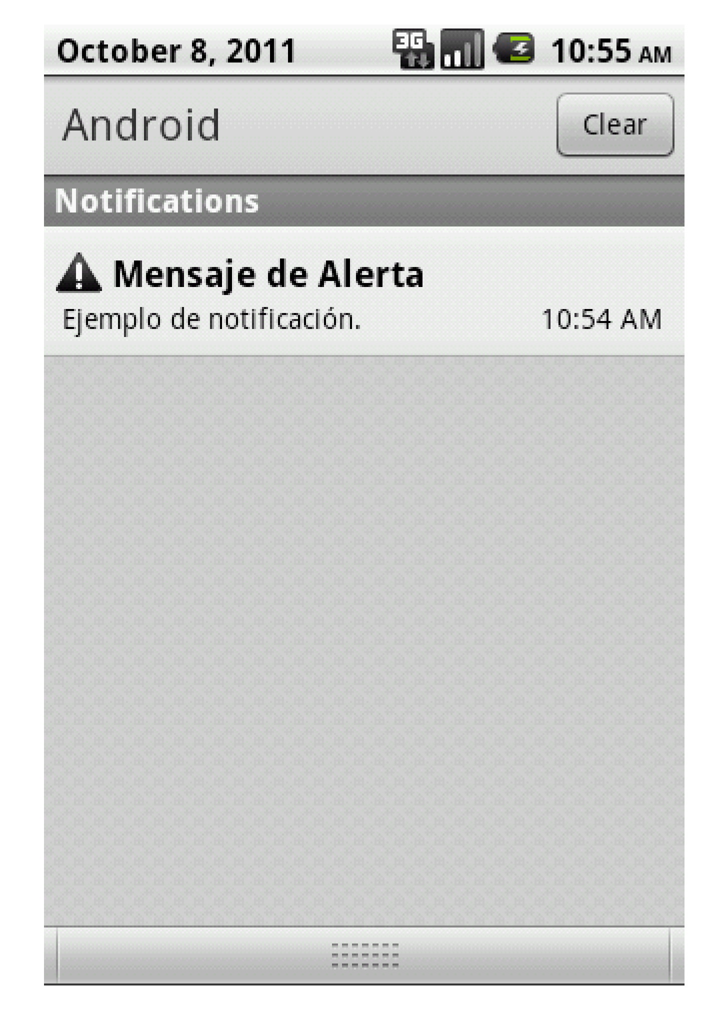
Por último, una vez tenemos completamente configuradas las opciones de nuestra notificación podemos generarla llamando al método `notify()`, pasando como parámetro un identificador único definido por nosotros y el objeto `Notification` construido.

```
//Enviar notificación
notManager.notify(NOTIF_ALERTA_ID, notif);
```

Ya estamos en disposición de probar nuestra aplicación de ejemplo. Para ello vamos a ejecutarla en el emulador de Android y pulsamos el botón que hemos implementado con todo el código anterior. Si todo va bien, debería aparecer en ese mismo momento nuestra notificación en la barra de estado, con el icono y texto definidos.



Si ahora salimos de la aplicación y desplegamos la bandeja del sistema podremos verificar el resto de información de la notificación tal como muestra la siguiente imagen:



Por último, si pulsamos sobre la notificación se debería abrir de nuevo automáticamente la aplicación de ejemplo. Además, la notificación también debería desaparecer de la bandeja del sistema, tal y como lo habíamos configurado en el código con la opción `FLAG_AUTO_CANCEL`.

En definitiva, como podéis comprobar es bastante sencillo generar notificaciones en la barra de estado de Android desde nuestras aplicaciones. Os animo a utilizar este mecanismo para notificar determinados eventos al usuario de forma bastante visual e intuitiva.

> El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-notificaciones.zip

Cuadros de Diálogo

Durante este curso ya hemos visto dos de las principales alternativas a la hora de mostrar notificaciones a los usuarios de nuestras aplicaciones: los *toast* y las *notificaciones de la barra de estado*. En este último apartado sobre notificaciones vamos a comentar otro mecanismo que podemos utilizar para mostrar o solicitar información puntual al usuario. El mecanismo del que hablamos son los *cuadros de diálogo*.

En principio, los *diálogos* de Android los podremos utilizar con distintos fines, en general:

- Mostrar un mensaje.
- Pedir una confirmación rápida.
- Solicitar al usuario una elección (simple o múltiple) entre varias alternativas.

De cualquier forma, veremos también cómo personalizar completamente un diálogo para adaptarlo a cualquier otra necesidad.

El uso de diálogos en Android guarda muchas semejanzas con el funcionamiento ya comentado de los menús, ya que se basa en la implementación de dos métodos de la propia actividad que los muestra, uno de ellos para crear el diálogo por primera vez, `onCreateDialog()`, y el segundo para poder modificarlos de forma dinámica cada vez que se muestran, `onPrepareDialog()`.

El primero de estos métodos recibe como parámetro el ID del diálogo que queremos crear. Podemos asignar estos ID a nuestro antojo pero deben identificar de forma única a cada diálogo de nuestra aplicación. La forma habitual de proceder será definir una constante en la actividad con algún nombre identificativo y utilizar ésta en el resto del código. Como en el caso de los menús, dentro de este método construiremos cada diálogo según el ID recibido y lo devolveremos como valor de retorno. Seguiríamos el siguiente esquema:

```
private static final int DIALOGO_TIPO_1 = 1;
private static final int DIALOGO_TIPO_2 = 2;
//...
protected Dialog onCreateDialog(int id) {
    Dialog dialogo = null;

    switch(id)
    {
        case DIALOGO_TIPO_1:
            dialogo = crearDialogo1();
            break;
        case DIALOGO_TIPO_2:
            dialogo = crearDialogo2();
            break;
        //...
        default:
            dialogo = null;
            break;
    }

    return dialogo;
}
```

La forma de construir cada diálogo dependerá de la información y funcionalidad que necesitemos. A continuación mostraré algunas de las formas más habituales.

Diálogo de Alerta

Este tipo de diálogo se limita a mostrar un mensaje sencillo al usuario, y un único botón de OK para confirma su lectura. Este tipo de diálogos los construiremos mediante la clase `AlertDialog`, y más concretamente su subclase `AlertDialog.Builder`. Su utilización es muy sencilla, bastará con crear un objeto de tipo `AlertDialog.Builder` y establecer

las propiedades del diálogo mediante sus métodos correspondientes: título [`setTitle()`], mensaje [`setMessage()`] y el texto y comportamiento del botón [`setPositiveButton()`]. Veamos un ejemplo:

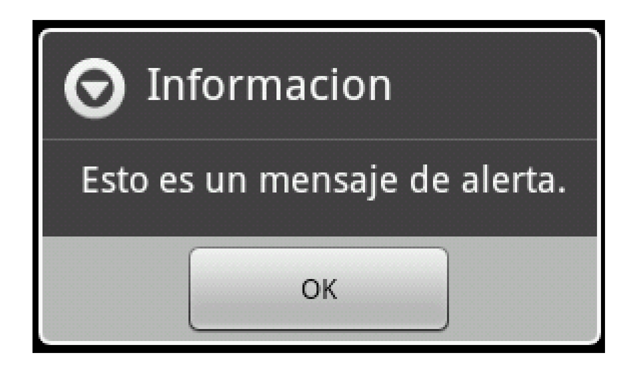
```
private Dialog crearDialogoAlerta()
{
    AlertDialog.Builder builder = new AlertDialog.Builder(this);

    builder.setTitle("Informacion");
    builder.setMessage("Esto es un mensaje de alerta.");
    builder.setPositiveButton("OK", new OnClickListener() {
        public void onClick(DialogInterface dialog, int which) {
            dialog.cancel();
        }
    });

    return builder.create();
}
```

Como vemos, al método `setPositiveButton()` le pasamos como argumentos el texto a mostrar en el botón, y la implementación del evento onClick en forma de objeto `OnClickListener`. Dentro de este evento, nos limitamos a cerrar el diálogo mediante su método `cancel()`, aunque podríamos realizar cualquier otra acción.

El aspecto de nuestro diálogo de alerta sería el siguiente:



Diálogo de Confirmación

Un diálogo de confirmación es muy similar al anterior, con la diferencia de que lo utilizaremos para solicitar al usuario que nos confirme una determinada acción, por lo que las posibles respuestas serán del tipo Sí/No.

La implementación de estos diálogos será prácticamente igual a la ya comentada para las alertas, salvo que en esta ocasión añadiremos dos botones, uno de ellos para la respuesta afirmativa (`setPositiveButton()`), y el segundo para la respuesta negativa (`setNegativeButton()`). Veamos un ejemplo:

```
private Dialog crearDialogoConfirmacion()
{
    AlertDialog.Builder builder = new AlertDialog.Builder(this);

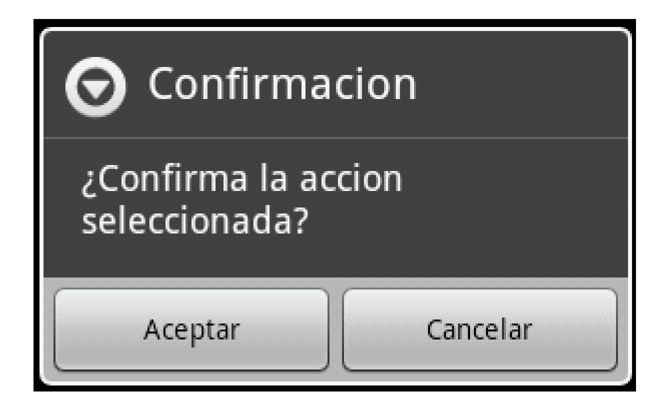
    builder.setTitle("Confirmacion");
    builder.setMessage("¿Confirma la accion seleccionada?");

    builder.setPositiveButton("Aceptar", new OnClickListener() {
```

```
    public void onClick(DialogInterface dialog, int which) {
        Log.i("Dialogos", "Confirmacion Aceptada.");
        dialog.cancel();
    }
    });
    builder.setNegativeButton("Cancelar", new OnClickListener() {
    public void onClick(DialogInterface dialog, int which) {
        Log.i("Dialogos", "Confirmacion Cancelada.");
        dialog.cancel();
    }
    });

    return builder.create();
}
```

En este caso, generamos a modo de ejemplo dos mensajes de log para poder verificar qué botón pulsamos en el diálogo. El aspecto visual de nuestro diálogo de confirmación sería el siguiente:



Diálogo de Selección

Cuando las opciones a seleccionar por el usuario no son sólo dos, como en los diálogos de confirmación, sino que el conjunto es mayor podemos utilizar los diálogos de selección para mostrar una lista de opciones entre las que el usuario pueda elegir.

Para ello también utilizaremos la clase `AlertDialog`, pero esta vez no asignaremos ningún mensaje ni definiremos las acciones a realizar por cada botón individual, sino que directamente indicaremos la lista de opciones a mostrar (mediante el método `setItems()`) y proporcionaremos la implementación del evento `onClick()` sobre dicha lista (mediante un listener de tipo `DialogInterface.OnClickListener`), evento en el que realizaremos las acciones oportunas según la opción elegida. La lista de opciones la definiremos como un array tradicional. Veamos cómo:

```
private Dialog crearDialogoSeleccion()
{
    final String[] items = {"Español", "Inglés", "Francés"};

    AlertDialog.Builder builder = new AlertDialog.Builder(this);

    builder.setTitle("Selección");


    builder.setItems(items, new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int item) {
```

```
            Log.i("Dialogos", "Opción elegida: " + items[item]);
        }
    });

    return builder.create();
}
```

En este caso el diálogo tendrá un aspecto similar a la interfaz mostrada para los controles Spinner.



Este diálogo permite al usuario elegir entre las opciones disponibles cada vez que se muestra en pantalla.

Pero, ¿y si quisiéramos recordar cuál es la opción u opciones seleccionadas por el usuario para que aparezcan marcadas al visualizar de nuevo el cuadro de diálogo? Para ello podemos utilizar los métodos setSingleChoiceItems() o setMultiChiceItems(), en vez de el setItems() utilizado anteriormente. La diferencia entre ambos métodos, tal como se puede suponer por su nombre, es que el primero de ellos permitirá una selección simple y el segundo una selección múltiple, es decir, de varias opciones al mismo tiempo, mediante controles CheckBox.

La forma de utilizarlos es muy similar a la ya comentada. En el caso de setSingleChoiceItems(), el método tan sólo se diferencia de setItems() en que recibe un segundo parámetro adicional que indica el índice de la opción marcada por defecto. Si no queremos tener ninguna de ellas marcadas inicialmente pasaremos el valor -1.

```
builder.setSingleChoiceItems(items, -1, new DialogInterface.OnClickListener()
{
    public void onClick(DialogInterface dialog, int item) {
        Log.i("Dialogos", "Opción elegida: " + items[item]);
    }
});
```

De esta forma conseguiríamos un diálogo como el de la siguiente imagen:



Si por el contrario optamos por la opción de selección múltiple, la diferencia principal estará en que tendremos que implementar un listener del tipo `DialogInterface.OnMultiChoiceClickListener`. En este caso, en el evento `onClick` recibiremos tanto la opción seleccionada (item) como el estado en el que ha quedado (`isChecked`). Además, en esta ocasión, el segundo parámetro adicional que indica el estado por defecto de las opciones ya no será un simple número entero, sino que tendrá que ser un array de booleanos. En caso de no querer ninguna opción seleccionada por defecto pasaremos el valor `null`.

```
builder.setMultiChoiceItems(items, null, new
DialogInterface.OnMultiChoiceClickListener() {
    public void onClick(DialogInterface dialog, int item, boolean isChecked) {
        Log.i("Dialogos", "Opción elegida: " + items[item]);
    }
});
```

Y el diálogo nos quedaría de la siguiente forma:



Tanto si utilizamos la opción de selección simple como la de selección múltiple, para salir del diálogo tendremos que pulsar la tecla "*Atrás*" de nuestro dispositivo, pero no perderemos el estado de las opciones. Si volvemos a abrir el diálogo éstas deberán continuar en el mismo

estado que la última vez que se cerró. Esto por supuesto se cumple mientras no se cierre la actividad asociada o se salga de la aplicación.

Y con esto terminamos con el apartado dedicado a notificaciones en Android. Hemos comentado los tres principales mecanismos de Android a la hora de mostrar mensajes y notificaciones al usuario (Toast, Barra de Estado, y Diálogos), todos ellos muy útiles para cualquier tipo de aplicación y que es importante conocer bien para poder decidir entre ellos dependiendo de las necesidades que tengamos.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-dialogos.zip



13 Depuración de aplicaciones

XIII. Depuración en Android

Logging en Android

Hacemos un pequeño alto en el camino en el Curso de Programación Android para hablar de un tema que, si bien no es específico de Android, sí nos va a resultar bastante útil a la hora de explorar otras características de la plataforma.

Una de las técnicas más útiles a la hora de depurar y/o realizar el seguimiento de aplicaciones sobre cualquier plataforma es la creación de logs de ejecución. Android por supuesto no se queda atrás y nos proporciona también su propio servicio y API de *logging* a través de la clase `android.util.Log`.

En Android, todos los mensajes de log llevarán asociada la siguiente información:

- Fecha/Hora del mensaje.
- Criticidad. Nivel de gravedad del mensaje (se detalla más adelante).
- PID. Código interno del proceso que ha introducido el mensaje.
- Tag. Etiqueta identificativa del mensaje (se detalla más adelante).
- Mensaje. El texto completo del mensaje.

De forma similar a como ocurre con otros frameworks de logging, en Android los mensajes de log se van a clasificar por su criticidad, existiendo así varias categorias (ordenadas de mayor a menor criticidad):

1. Error
2. Warning
3. Info
4. Debug
5. Verbose

Para cada uno de estos tipos de mensaje existe un método estático independiente que permite añadirlo al log de la aplicación. Así, para cada una de las categorías anteriores tenemos disponibles los métodos `e()`, `w()`, `i()`, `d()` y `v()` respectivamente.

Cada uno de estos métodos recibe como parámetros la etiqueta (*tag*) y el texto en sí del mensaje. Como etiqueta de los mensajes, aunque es un campo al que podemos pasar cualquier valor, suele utilizarse el nombre de la aplicación o de la actividad concreta que genera el mensaje. Esto nos permitirá más tarde crear filtros personalizados para identificar y poder visualizar únicamente los mensajes de log que nos interesan, entre todos los generados por Android [que son muchos] durante la ejecución de la aplicación.

Hagamos un miniprograma de ejemplo para ver cómo fuenciona esto. El programa será tan simple como añadir varios mensajes de log dentro del mismo onCreate de la actividad principal y ver qué ocurre. Os muestro el código completo:

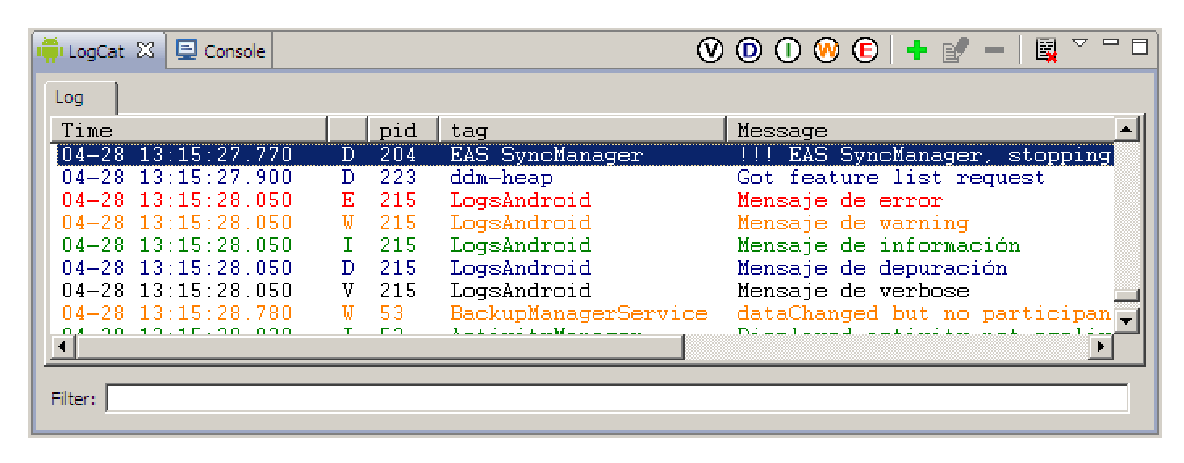
```
public class LogsAndroid extends Activity {

    private static final String LOGTAG = "LogsAndroid";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        Log.e(LOGTAG, "Mensaje de error");
        Log.w(LOGTAG, "Mensaje de warning");
        Log.i(LOGTAG, "Mensaje de información");
        Log.d(LOGTAG, "Mensaje de depuración");
        Log.v(LOGTAG, "Mensaje de verbose");
    }
}
```

Si ejecutamos la aplicación anterior en el emulador veremos cómo se abre la pantalla principal que crea Eclipse por defecto y aparentemente no ocurre nada más. ¿Dónde podemos ver los mensajes que hemos añadido al log? Pues para ver los mensajes de log nos tenemos que ir a la perspectiva de Eclipse llamada *DDMS*. Una vez en esta perspectiva, podemos acceder a los mensajes de log en la parte inferior de la pantalla, en una vista llamada *LogCat*. En esta ventana se muestran todos los mensajes de log que genera Android durante la ejecución de la aplicación, que son muchos, pero si buscamos un poco en la lista encontraremos los generados por nuestra aplicación, tal como se muestra en la siguiente imagen:



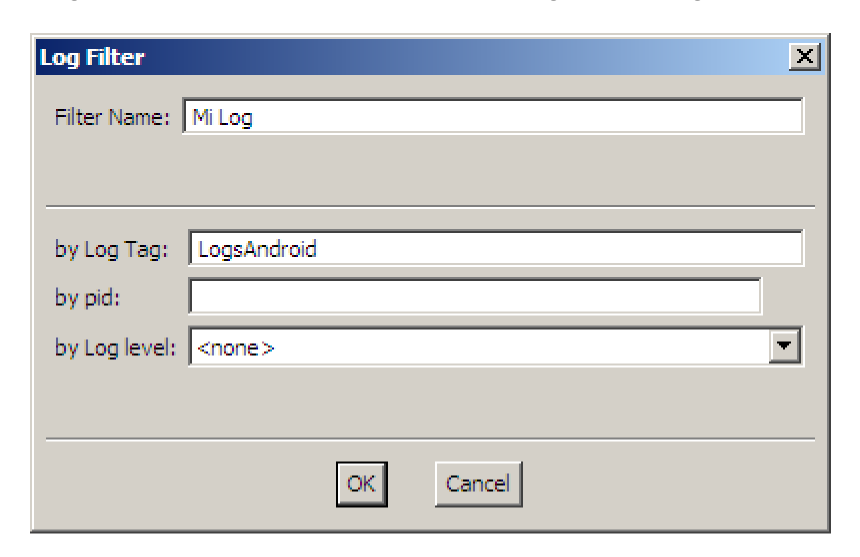
Como se puede observar, para cada mensaje se muestra toda la información que indicamos al principio del apartado, además de estar diferenciados por un color distinto según su criticidad.

En este caso de ejemplo, los mensajes mostrados son pocos y fáciles de localizar en el log, pero para una aplicación real, el número de estos mensajes puede ser mucho mayor y aparecer intercalados caóticamente entre los demás mensajes de Android. Para estos casos, la ventada de LogCat ofrece una serie de funcionalidades para facilitar la visualización y búsqueda de determinados mensajes.

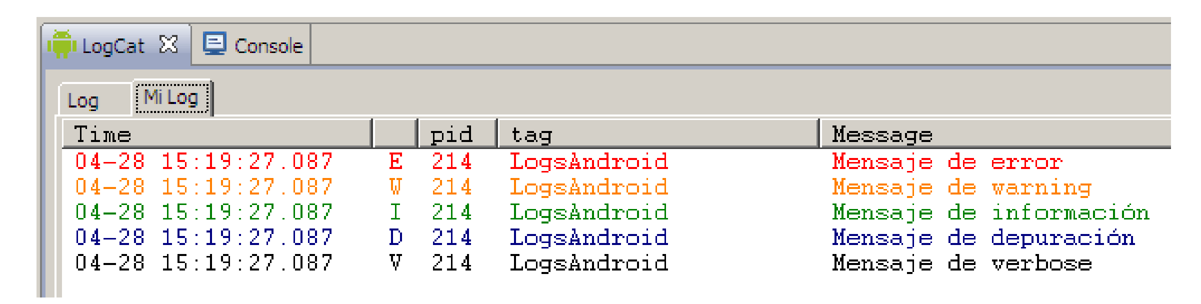
Por ejemplo, podemos restringir la lista para que sólo muestre mensajes con una determinada criticidad mínima. Esto se consigue pulsando alguno de los 5 primeros botones que se observan en la parte superior derecha de la ventana de log. Así, si por ejemplo pulsamos sobre el botón de la categoría *Info* (en verde), en la lista sólo se mostrarán los mensajes con criticidad *Error*, *Warning* e *Info*.

Otro método de filtrado más interesante es la definición de filtros personalizados (botón "+" verde), donde podemos filtrar la lista para mostrar únicamente los mensajes con un PID o Tag determinado. Si hemos utilizado como etiqueta de los mensajes el nombre de nuestra aplicación o de nuestras actividades esto nos proporcionará una forma sencilla de visualizar sólo los mensajes generados por nuestra aplicación.

Así, para nuestro ejemplo, podríamos crear un filtro indicando como Tag la cadena "*LogsAndroid*", tal como se muestra en la siguiente imagen:



Esto creará una nueva ventana de log con el nombre que hayamos especificado en el filtro, donde sólo aparecerán nuestros 5 mensajes de log de ejemplo:



Por último, cabe mencionar que existe una variante de cada uno de los métodos de la clase Log que recibe un parámetro más en el que podemos pasar un objeto de tipo excepción. Con esto conseguimos que, además del mensaje de log indicado, se muestre también la traza de error generada con la excepción.

Veamos esto con un ejemplo, y para ello vamos a forzar un error de división por cero, vamos a capturar la excepción y vamos a generar un mensaje de log con la variante indicada:

```
try
{
    int a = 1/0;
}
catch(Exception ex)
{
    Log.e(LOGTAG, "División por cero!", ex);
}
```

Si volvemos a ejecutar la aplicación y vemos el log generado, podremos comprobar cómo se muestra la traza de error correspondiente generada con la excepción.

```
Message
Mensaje de error
Mensaje de warning
Mensaje de información
Mensaje de depuración
Mensaje de verbose
División por cero!
java.lang.ArithmeticException: divide by zero
    at net.sgoliver.android.LogsAndroid.onCreate(LogsAndroid.java:24)
    at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1047)
    at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2459)
    at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2512)
    at android.app.ActivityThread.access$2200(ActivityThread.java:119)
    at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1863)
```

En definitiva, podemos comprobar como la generación de mensajes de log puede ser una herramienta sencilla pero muy efectiva a la hora de depurar aplicaciones que no se ajustan mucho a la depuración paso a paso, o simplemente para generar trazas de ejecución de nuestras aplicaciones para comprobar de una forma sencilla cómo se comportan.

El código fuente completo de este apartado está disponible entre los ejemplos suministrados con el manual. **Fichero:** /Codigo/android-logging.zip